

# Visual Inertial SLAM auf Android Geräten für Augmented Reality

BACHELORARBEIT

ausgearbeitet von

Jannis Malte Möller

zur Erlangung des akademischen Grades  
BACHELOR OF SCIENCE (B.Sc.)

vorgelegt an der

TECHNISCHEN HOCHSCHULE KÖLN  
CAMPUS GUMMERSBACH  
FAKULTÄT FÜR INFORMATIK UND  
INGENIEURWISSENSCHAFTEN

im Studiengang

INFORMATIK

Erster Prüfer: Prof. Dr. Martin Eisemann  
Technische Hochschule Köln

Zweiter Prüfer: Prof. Dr. Heiner Klocke  
Technische Hochschule Köln

Gummersbach, im Februar 2018

**Adressen:** Jannis Malte Möller  
Postfach 10 06 07  
51606 Gummersbach  
jmoeller@th-koeln.de

Prof. Dr. Martin Eisemann  
Technische Hochschule Köln  
Institut für Informatik  
Steinmüllerallee 1  
51643 Gummersbach  
martin.eisemann@th-koeln.de

Prof. Dr. Heiner Klocke  
Technische Hochschule Köln  
Institut für Informatik  
Steinmüllerallee 1  
51643 Gummersbach  
heinrich.klocke@th-koeln.de

# Kurzfassung

**Visual Inertial SLAM**-Algorithmen können in der **Augmented Reality** für die Erkennung und das Tracking der Umgebung genutzt werden.

Im Rahmen des Projekts Augmented Reality Campus habe ich das iOS-Open-Source-Projekt VINS-Mobile (Li et al., 2017) auf die **Android**-Plattform portiert. Den theoretischen Ansatz des zugrundeliegenden Algorithmus stelle ich vor. Im Anschluss an die Dokumentation der Portierung vergleiche ich die beiden Plattformen in mehreren Performance-Untersuchungen. Die Ergebnisse verwende ich, um die Frage zu beantworten, ob die Portierung für den Einsatz im AR Campus-Projekt geeignet ist.

Dabei komme ich zu dem Schluss, dass die Android-Portierung in Funktionsumfang und Qualität der iOS-Version fast gleichzusetzen ist. Welche Arbeitsschritte bis zur Nutzbarkeit im Projekt AR Campus noch nötig sind, beschreibe ich am Ende der Arbeit.

# Abstract

In **Augmented Reality Visual inertial SLAM** algorithms can be used for detection and tracking of the environment.

As part of the Augmented Reality Campus project, I have ported the iOS open-source project VINS-Mobile (Li et al., 2017) to the **Android** platform. I present the theoretical approach of the underlying algorithm. Following the documentation of the porting process, I compare the two platforms in several performance tests. I use the results to answer the question whether the port is suitable for use in the AR Campus project.

I come to the conclusion that the Android port is almost equal in functionality and quality compared to the iOS version. At the end of the work, I describe which work steps need to be completed for the port to be usable in the AR Campus project.

# Inhaltsverzeichnis

<b>1. Einleitung</b>	<b>1</b>
<b>2. Stand der Technik</b>	<b>2</b>
<b>3. Überblick</b>	<b>3</b>
<b>4. VINS Framework</b>	<b>4</b>
4.1. Sliding Window . . . . .	5
4.2. Measurement Pre-Processing . . . . .	6
4.2.1. Feature Detection and Tracking . . . . .	6
4.2.2. IMU Pre-integration . . . . .	6
4.3. Initialization . . . . .	8
4.3.1. Vision-Only Structure from Motion in Sliding Window . . . . .	8
4.3.2. Visual-Inertial Alignment . . . . .	9
4.3.3. Termination Criteria . . . . .	10
4.4. Tightly-Coupled Nonlinear Optimization . . . . .	11
4.4.1. Visual Inertial Bundle Adjustment Formulation . . . . .	11
4.4.2. IMU Measurement Model . . . . .	12
4.4.3. Visual Measurement Model . . . . .	12
4.4.4. Loop Closure Model . . . . .	13
4.4.5. Marginalization . . . . .	14
4.5. Loop Closure . . . . .	15
4.5.1. Keyframe Database . . . . .	16
4.5.2. Loop Detection . . . . .	16
4.5.3. Relocalization . . . . .	17
4.6. Eignung für mobile Geräte . . . . .	18
<b>5. Portierung</b>	<b>19</b>
5.1. Externe Bibliotheken . . . . .	19
5.1.1. Boost . . . . .	20
5.1.2. Eigen . . . . .	21
5.1.3. OpenCV . . . . .	21
5.1.4. Ceres Solver . . . . .	22
5.1.5. DBoW2 . . . . .	23
5.1.6. GNU C++ Library . . . . .	24
5.2. Übersetzung . . . . .	24
5.2.1. Herangehensweise . . . . .	24
5.2.2. Objective C Dateien . . . . .	25
5.2.3. Android-Applikationsaufbau . . . . .	27
5.2.4. Vergleich zwischen iOS und Android . . . . .	32



## *Inhaltsverzeichnis*

5.2.5. Probleme . . . . .	33
5.3. Kalibrierung . . . . .	34
5.3.1. Intrinsische Parameter . . . . .	34
5.3.2. Extrinsische Parameter . . . . .	36
<b>6. Ergebnisse</b>	<b>38</b>
6.1. Test-Hardware . . . . .	38
6.2. Performance auf Android Geräten . . . . .	39
6.2.1. Vor Compiler-Optimierungen . . . . .	40
6.2.2. Nach Compiler-Optimierungen . . . . .	40
6.3. Experimentelle Ergebnisse . . . . .	42
6.3.1. Versuchsbedingungen . . . . .	42
6.3.2. Indoor Experiment . . . . .	43
6.3.3. Outdoor Experiment . . . . .	45
<b>7. Diskussion</b>	<b>47</b>
7.1. Ergebnis-Bewertung . . . . .	47
7.2. Anforderungen durch AR-Campus . . . . .	47
<b>8. Zusammenfassung und Ausblick</b>	<b>50</b>
<b>9. Persönliche Entwicklung</b>	<b>51</b>
<b>Abbildungsverzeichnis</b>	<b>I</b>
<b>Tabellenverzeichnis</b>	<b>II</b>
<b>Literaturverzeichnis</b>	<b>IV</b>
<b>Anhang A. Performance Statistiken</b>	<b>V</b>
A.1. Vor Compiler-Optimierung . . . . .	V
A.2. Nach Compiler-Optimierung . . . . .	VI
<b>Anhang B. Digitale Anlagen</b>	<b>VIII</b>

# 1. Einleitung

**Thema** *Visual Inertial Simultaneous Localising and Mapping* beschreibt eine Klasse von Algorithmen, die sowohl visuelle als auch inertielle Messdaten verwenden, um sich in der Umgebung zu lokalisieren und diese gleichzeitig zu kartografieren. Neben dem Forschungsgebiet der Robotik sind sie auch für die *Augmented Reality* (AR) von Interesse. Hier ist die korrekte Darstellung einer 3D-Szene in einem Echtzeit-Kamerabild der realen Welt das Ziel. Das Forschungsgebiet ist sehr aktiv und es werden immer wieder neue Ansätze veröffentlicht.

Gleichzeitig wird die Hardware immer leistungsfähiger. Gerade der Smartphone-Markt entwickelt sich rasend schnell weiter. So kommt es, dass Algorithmen, die vor einigen Jahren noch Desktop-Hardware erforderten, inzwischen für den Einsatz auf mobilen Endgeräten in Frage kommen.

Das Projekt Augmented Reality Campus der Technischen Hochschule Köln Campus Gummersbach soll den Studenten in besonderer Form Lerninhalte vermitteln und ortsbezogene Informationen bereitstellen können. Die erste angestrebte Zielplattform ist *Android*.

Die Bachelorarbeit befasst sich im Rahmen dieses Projekts mit folgender Problem-, Ziel- und Fragestellung:

**Problemstellung** Um auf den Endgeräten der Studenten ein möglichst robustes und somit immersives Augmented Reality-Erlebnis zu ermöglichen, wird ein dreidimensionales Lokalisieren innerhalb der Umgebung benötigt. Die meisten verfügbaren Augmented Reality Frameworks bieten allerdings nur das Tracking von bestimmten Markern an. Da alle gängigen Smartphones heutzutage mit einer Kamera und einem Bewegungssensor ausgestattet sind, bietet sich für eine bessere Lösung die Visual Inertial Odometry an.

**Zielstellung** Das Ziel dieser Bachelorarbeit soll es sein, das für iOS entwickelte Open-Source Projekt VINS-Mobile (Li et al., 2017) auf die Android-Plattform zu portieren. Dabei müssen die an die Zielplattform Android gekoppelten Soft- und Hardwarebedingungen untersucht und berücksichtigt werden. Auch wenn sich die Hardware immer weiterentwickelt, sind Prozessorleistung und Speicher immer noch ein begrenzender Faktor. Dieses Projekt wurde gewählt, da es eines der wenigen ist, das mobile Hardware nutzt und auf iOS vielversprechende Ergebnisse liefert.

**Fragestellung** An den praktischen Teil anschließend sollen die folgenden Fragestellungen beantwortet werden: Wie gut performt die Android-Portierung verglichen mit der iOS-Version und dem Google Tango AR Framework? Und ist sie für den späteren Einsatz im Projekt AR Campus geeignet?

## 2. Stand der Technik

Im Bereich der Augmented Reality-Anwendungen und -Frameworks gibt es sehr viele verschiedene Lösungen. Der Favorit für die Verwendung im Projekt AR-Campus ist das OpenSource AR-Framework *ARToolKit*<sup>1</sup>. Durch die Kompatibilität mit Unity bietet es die Möglichkeit, eine einzige Anwendung für mehrere Plattformen gleichzeitig zu entwickeln. Es bietet in der aktuellen Version allerdings keinerlei Möglichkeit, eine 3D-Umgebung anhand von Feature-Matching zu erkennen oder zu tracken. Es beschränkt sich ausschließlich auf 2D-Bilder bzw. Marker. Mit zukünftigen Versionen dieses Frameworks sollen verbessertes Feature Tracking und Flächenerkennung ermöglicht werden. Doch die Weiterentwicklung scheint im letzten Jahr zum Erliegen gekommen zu sein. Andere Open-Source-Lösungen sind in Umfang und Funktion noch stärker beschränkt, einige bieten nicht mehr als eine Abstraktionsschicht um die Bildverarbeitungsbibliothek *OpenCV*<sup>2</sup>.

Weiterhin sind einige proprietäre Lösungen verfügbar, die jedoch nicht quelloffen und meistens auch mit hohen Kosten in der späteren Nutzung verbunden sind. Sie kommen daher nicht für den Einsatz in dem Projekt in Frage. Zu ihnen gehören unter anderem *Vuforia*<sup>3</sup>, *Kudan*<sup>4</sup> und *MAXST*<sup>5</sup>.

Im Forschungsbereich der Visual Inertial Odometry und der SLAM-Algorithmen gibt es viele Open-Source-Lösungen. Der Großteil dieser ist allerdings für andere Plattformen mit anderen Hardware-Voraussetzungen konzipiert bzw. implementiert. Oft werden die Berechnungen offline, also nicht in Echtzeit oder auf leistungsfähiger Desktop-Hardware ausgeführt.

Eine der wenigen mobilen Lösungen ist das Projekt *VINS-Mono* (Qin et al., 2017) bzw. die iOS-Portierung *VINS-Mobile* (Li et al., 2017). Der Quell-Code sowie die zugrundeliegenden wissenschaftlichen Arbeiten sind in dem GitHub Repository <https://github.com/HKUST-Aerial-Robotics/VINS-Mobile> veröffentlicht.

---

<sup>1</sup><https://www.artoolkit.org/>

<sup>2</sup><https://opencv.org/>

<sup>3</sup><https://www.vuforia.com/>

<sup>4</sup><https://www.kudan.eu/>

<sup>5</sup><http://maxst.com/>

### 3. Überblick

In diesem Kapitel wird ein Überblick über die Teile der Arbeit gegeben. Die Arbeit ist im Hauptteil in drei große Abschnitte gegliedert.

Der erste Abschnitt entspricht **Kapitel 4**. In diesem Kapitel wird der dem VINS Framework zu Grunde liegende Algorithmus erläutert. Die einzelnen Schritte werden detailliert erklärt und es wird darauf eingegangen, wieso diese Algorithmus-Pipeline für den Einsatz auf mobilen Geräten gut geeignet ist.

Abschnitt zwei besteht aus **Kapitel 5**. Hier werden die Schritte vorgestellt, die den praktischen Teil der Arbeit umfassen. Dazu gehört die Einbindung der verwendeten externen Bibliotheken und die Herangehensweise und Ergebnisse der Übersetzung des iOS spezifischen Codes. Zuletzt wird noch der Vorgang zur Kalibrierung der Test-Hardware beschrieben.

Der dritte Abschnitt umfasst die Kapitel 6 und 7.

In **Kapitel 6** werden die experimentellen Ergebnisse der Portierung dargestellt. Untergliedert wird dieses Kapitel in die quantitative Untersuchung der Performance auf der Android Test-Hardware. Außerdem wird eine qualitative Gegenüberstellung mit der ursprünglichen Implementierung auf iOS und dem Augmented Reality-Projekt Google Tango sowohl innerhalb als auch außerhalb eines Gebäudes durchgeführt.

Im Anschluss an die Untersuchungen werden die Ergebnisse in **Kapitel 7** diskutiert und es wird auf die Möglichkeiten und Einschränkungen in dem angestrebten Praxis-kontext eingegangen.

Zuletzt fasst **Kapitel 8** die Inhalte dieser Arbeit noch einmal zusammen und gibt einen Ausblick auf eine mögliche zukünftige Fortführung. In **Kapitel 9** reflektiere ich zum Abschluss meine Erfahrungen, die ich aus diesem Projekt gewonnen habe.

## 4. VINS Framework

In diesem Kapitel wird der in dem Framework VINS-Mobile implementierte und in Li et al. (2017) vorgestellte Algorithmus in seinen einzelnen Schritten erklärt.

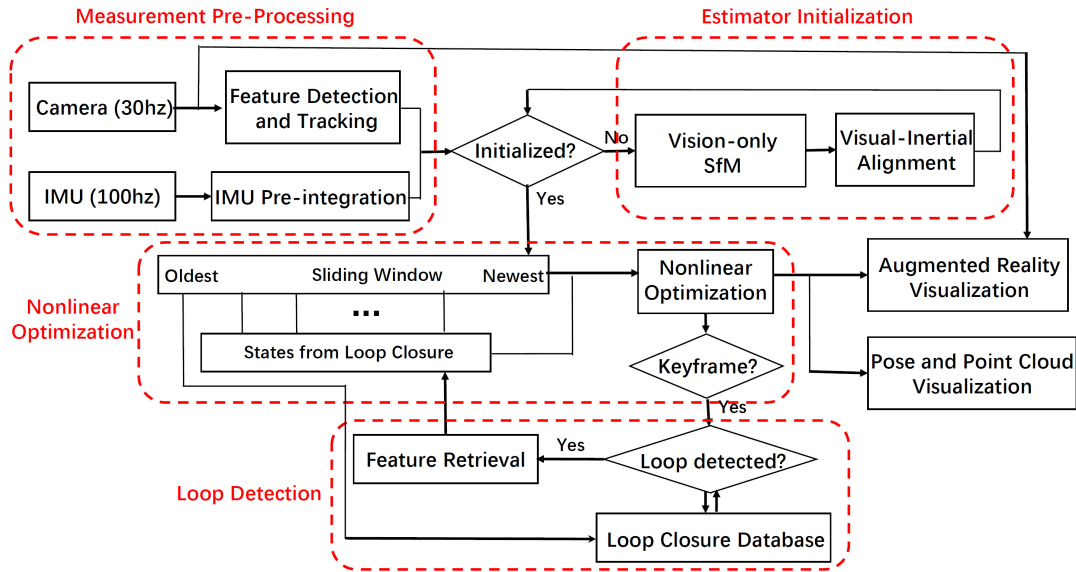


Abbildung 4.1.: Blockdiagramm der gesamten Pipeline mit Zusammenhängen der einzelnen Bestandteile (nach Li et al., 2017, S. 2)

Zunächst einmal wird in Abschnitt 4.1 die allgegenwärtige Datenstruktur des „Sliding Window“ beschrieben.

Die darauf folgenden Abschnitte erklären die vier in Abbildung 4.1 rot umrandeten Blöcke im Detail. Abschnitt 4.2 erläutert die Vorverarbeitung der Messungen der Kamera und der inertialen Messeinheit. Der Initialisierungsvorgang wird in Abschnitt 4.3 behandelt. In Abschnitt 4.4 wird die eng geknüpfte und nicht lineare Optimierung erklärt und zuletzt wird in Abschnitt 4.5 die Loop-Erkennung und -Schließung erläutert.

### 4.1. Sliding Window

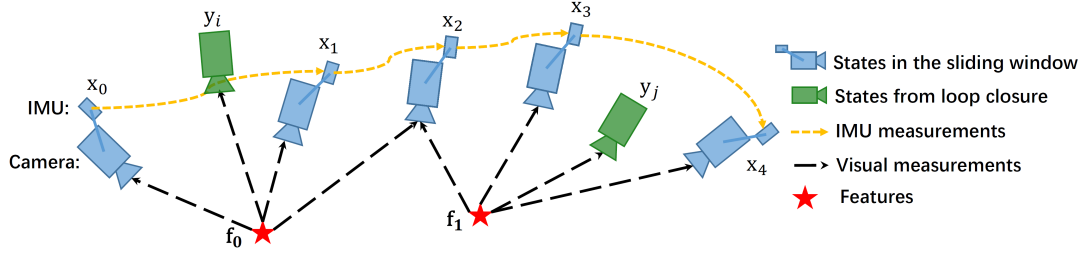


Abbildung 4.2.: Sliding Window Formulation (Li et al., 2017, S. 3)

Dieser Abschnitt stellt die überall im Algorithmus verwendete Struktur des Sliding Window vor. Wie der Name vermuten lässt, handelt es sich dabei um ein Betrachtungsfenster, das immer nur eine begrenzte Anzahl an Daten hält, bevor es weiter verschoben wird. In diesem Fall sind die Daten die Zustände zu den Zeiten der Kamera-Keyframes. Welche Daten diese Zustände umfassen, wird in Gleichung (4.1) formuliert. Außerdem werden in dem Fenster auch die in den Frames beobachteten Features gespeichert.

Eine Visualisierung dieser Formulierung kann in Abbildung 4.2 betrachtet werden. Neben den eben beschriebenen Daten sind dort auch zusätzliche Zustände von außerhalb des Sliding Window grün dargestellt. Sie stammen aus der Keyframe-Datenbank und dienen der Loop-Erkennung und -Schließung. Mehr dazu in Abschnitt 4.5.

Der Zustand des Sliding Window kann in der folgenden Formulierung als Vektor zusammengefasst werden:

$$\begin{aligned} \mathcal{X} &= [x_0, x_1, \dots, x_n, \lambda_0, \lambda_1, \dots, \lambda_m] \\ x_k &= [\mathbf{p}_{b_k}^w, \mathbf{v}_{b_k}^w, \mathbf{q}_{b_k}^w, \mathbf{b}_a, \mathbf{b}_g], k \in [0, n] \end{aligned} \quad (4.1)$$

(Li et al., 2017, S. 3)

Hierbei sind  $x_k$  die einzelnen Keyframe-Zustände und  $n$  deren Anzahl.  $\lambda$  sind die Features mit Referenz zu dem Frame, in dem sie das erste Mal auftraten. Gespeichert werden sie als Inverses der Tiefe in diesem Frame.  $m$  ist die Anzahl der aktuellen Features. Die Zustände  $x_k$  setzen sich zusammen aus der Position  $\mathbf{p}_{b_k}^w$ , der Geschwindigkeit  $\mathbf{v}_{b_k}^w$  und der Orientierung  $\mathbf{q}_{b_k}^w$  im World-Frame. Außerdem gehören noch das Bias des Beschleunigungssensors  $\mathbf{b}_a$  und des Gyroskops  $\mathbf{b}_g$  dazu.

Der Übersichtlichkeit halber wird an dieser Stelle die Transformation von Kamera-Frame zum Frame der inertialen Messeinheit (IMU) ausgelassen. Genauere Informationen, wie diese Daten ermittelt werden, liefern die folgenden Kapitel.

## 4.2. Measurement Pre-Processing

Die Vorverarbeitung der Messungen ist der erste Schritt in der Pipeline. Sie wird benötigt, damit die rohen Kamera- und IMU-Daten in die Pipeline der *Estimator Initialization* bzw. der *Nonlinear Optimization* eingespeist und verwendet werden können.

Aufgeteilt ist dieser Schritt in die Verarbeitung der Kamerabilder mittels Feature-Erkennung und Tracking (Abschnitt 4.2.1) und die *Pre-Integration* der IMU-Messdaten (Abschnitt 4.2.2).

### 4.2.1. Feature Detection and Tracking

Dieser Unterabschnitt stellt die verwendeten Feature-Erkennungs- und Tracking-Algorithmen vor. Außerdem werden die Kriterien für die Auswahl von Keyframes erklärt.

Neue Features werden mit dem FAST corner detector (Rosten and Drummond, 2006) erkannt. Um eine möglichst gleichmäßige Verteilung zu gewährleisten, wird ein minimaler Abstand der Features von 30 Pixeln festgelegt. Für das Tracking der Features wird der KLT-Feature-Tracker verwendet. Dieser nutzt Informationen aus der Berechnung des Optical Flow, um effizient die gesuchten Features wiederzuerkennen.

Die Kamera nimmt Bilder mit einer Frequenz von 30 Hz auf. Für jeden dieser Frames wird das Tracking schon bekannter Features durchgeführt. Aus Performance-Gründen wird die Erkennung neuer Features aber nur bei jedem dritten Bild, also mit 10 Hz durchgeführt. Auch aus Performance-Gründen werden maximal 60 neue Features ermittelt.

Neben dem Erkennen und Tracken der Features wird auch die Keyframe-Auswahl bereits in diesem Schritt getroffen. Denn nicht alle Frames der Kamera sind auch für die weitere Verarbeitung geeignet. Informationen über die räumliche Tiefe der Features können bei einem monokularen Algorithmus nämlich nur aus Bewegungen extrahiert werden, die eine ausreichend große Parallaxe aufweisen. Deswegen werden neue Keyframes nur dann ausgewählt, wenn der durchschnittliche Parallax-Effekt der wiedererkannten Features über einem bestimmten Grenzwert liegt.

Die direkte Pixel-Verschiebung der Features auf der Bildebene tritt auch bei einer reinen Rotationsbewegung auf. Da hieraus aber keine Tiefeninformationen gewonnen werden können, werden die IMU-Messungen zum Kompensieren der Rotation für die Berechnung des Parallaxen-Werts hinzugezogen.

Ein zusätzliches Kriterium für die Wahl ist, ob eine bestimmte Anzahl alter Features nicht mehr erkannt wird. Auch dies spricht dafür, dass eine ausreichend große Bewegung aufgetreten ist.

### 4.2.2. IMU Pre-integration

Dieser Unterabschnitt behandelt die Vorverarbeitung der Bewegungssensor-Daten. Die inertiale Messeinheit (IMU) kann mit einer wesentlich höheren Frequenz (100Hz) als die Kamera neue Messdaten zur Verfügung stellen. Aus diesem Grund werden die Messungen zwischen den einzelnen Kamerabildern integriert und als Blöcke behandelt. Dies hat den positiven Effekt, dass durch eine Vereinfachung der mathematischen Komplexität in den folgenden Schritten Prozessorleistung gespart werden kann. Au-

#### 4. VINS Framework

ßerdem ermöglicht es die Nutzung der IMU-Messungen ohne den Ausgangszustand von Geschwindigkeit und Orientierung in der Welt zu kennen.

Die Qualität von günstigen, in Smartphones verbauten IMU-Sensoren ist meistens schlecht. Um trotzdem möglichst genaue Ergebnisse zu erzielen, wird in dieser Implementierung das *Bias* und *Noise* des Sensors berücksichtigt. Das Bias kann als Offset des durchschnittlichen Messwerts von dem tatsächlichen Wert und das Noise als Rauschen der Messwerte um diesen Offset herum angesehen werden.

Die folgenden Gleichungen stellen dar, wie sich die rohen Messwerte  $\hat{\omega}^b$  des Gyroskops und  $\hat{\mathbf{a}}^b$  aus diesen Werten zusammensetzen:

$$\begin{aligned}\hat{\omega}^b &= \omega^b + \mathbf{b}_g + \boldsymbol{\eta}_g \\ \hat{\mathbf{a}}^b &= \mathbf{q}_w^b (\mathbf{a}^w + \mathbf{g}^w) + \mathbf{b}_a + \boldsymbol{\eta}_a\end{aligned}\tag{4.2}$$

(Li et al., 2017, S. 4)

$\omega^b$  bezeichnet die tatsächliche lokale Winkelgeschwindigkeit. Die Beschleunigung  $\mathbf{a}^w$  ist zusammen mit dem Gravitationsvektor  $\mathbf{g}^w$  im World-Frame angegeben.  $\mathbf{q}_w^b$  ist die Rotationsmatrix, die die Orientierung dieser beiden Vektoren in den lokalen IMU-Frame transformiert.  $\mathbf{b}$  und  $\boldsymbol{\eta}_a$  sind Bias und Noise des Gyroskop bzw. des Accelerometers.

Die Messungen  $\hat{\mathbf{a}}$  und  $\hat{\omega}$  werden verwendet, um die Pre-Integration nach der folgenden Formel iterativ zu berechnen: Die Formulierung folgt den Sätzen der gleichmäßig beschleunigten Bewegung für das jeweiligen Intervall  $[i, i + 1]$

$$\begin{aligned}\hat{\alpha}_{i+1}^{b_k} &= \hat{\alpha}_i^{b_k} + \hat{\beta}_i^{b_k} \delta t + \frac{1}{2} \mathbf{R}(\hat{\gamma}_i^{b_k})(\hat{\mathbf{a}}_i - \mathbf{b}_{a_i}) \delta t^2 \\ \hat{\beta}_{i+1}^{b_k} &= \hat{\beta}_i^{b_k} + \mathbf{R}(\hat{\gamma}_i^{b_k})(\hat{\mathbf{a}}_i - \mathbf{b}_{a_i}) \delta t \\ \hat{\gamma}_{i+1}^{b_k} &= \hat{\gamma}_i^{b_k} \otimes \left[ \frac{1}{2} (\hat{\omega}_i - \mathbf{b}_{w_i}) \delta t \right]\end{aligned}\tag{4.3}$$

(Qin et al., 2017, S. 4)

$i$  liegt in den Grenzen der Keyframes  $b_k$  und  $b_k + 1$  und beschreibt den Zeitpunkt der einzelnen IMU-Messungen.  $\delta t$  gibt das Zeitintervall zwischen  $i$  und  $i + 1$  an.  $\mathbf{R}(\boldsymbol{\gamma})$  ist die Rotationsmatrix aus dem Quaternion  $\boldsymbol{\gamma}$ . Die Ergebnisse der Iterationsschritte sind die folgenden:

$\hat{\alpha}^{b_k}$	Position
$\hat{\beta}^{b_k}$	Geschwindigkeit
$\hat{\gamma}^{b_k}$	Rotation als Quaternion

Das Zirkumflex ( $\hat{\cdot}$ ) macht bei diesen drei Werten deutlich, dass es sich um die Resultate der Pre-Integration handelt.

Zu Beginn sind  $\hat{\alpha}_i^{b_k}$ ,  $\hat{\beta}_i^{b_k}$  und  $\hat{\gamma}_i^{b_k}$  mit  $i$  an der unteren Grenze  $b_k$  noch unbekannt und werden als 0 bzw. Identitäts-Quaternion behandelt. Das gleiche gilt für das Bias



und das Noise. In den folgenden Schritten wird es geschätzt und in dem nicht linearen Optimierungsvorgang immer wieder an die gegebenen Messungen angepasst.

Es wird immer nur der lokale Body-Frame als Referenz-Frame betrachtet, sodass die Pre-Integration unabhängig vom vorherigen Zustand durchgeführt werden kann.

### 4.3. Initialization

In diesem Abschnitt geht es um die zeitlich erste Phase des Algorithmus, in der die Geschwindigkeit, die Ausrichtung und die metrische Skalierung der Umgebung initialisiert wird.

Algorithmen, die ausschließlich auf monokularen Kamerabildern operieren, haben ohne weitere Vorannahmen keine Möglichkeit, die Skalierung der beobachteten Welt zu erkennen. Außerdem haben sie auch keine Referenz der Orientierung. Um diese Informationen bestimmen zu können, müssen die IMU-Messungen hinzugezogen werden.

Für gewöhnlich wird dazu eine Vorgehensweise gewählt, bei der das System in der Initialisierungsphase ruhig gehalten wird. Unter der Annahme, dass es stationär ist, wird dann der Durchschnitt der IMU-Messung der ersten Sekunden als Gravitationsvektor festgelegt. Die anfänglichen Posen werden dann direkt aus den Messungen der IMU berechnet.

Das Problem bei dieser Herangehensweise ist zum einen, dass das Bias der verwendeten Smartphone-IMUs die Genauigkeit stark negativ beeinflussen kann. Zum anderen führt das Auftreten von Bewegung in der Initialisierungsphase des Vorgangs zu einem falschen Ergebnis.

Im VINS-Framework kommt daher als robuste Initialisierungsprozedur eine lose gekoppelte Ausrichtung der IMU-Pre-Integrations-Daten mit den Outputs eines Vision-Only Structure from Motion (SfM)-Algorithmus zum Einsatz.

Zunächst wird über den Vision-Only SfM-Algorithmus eine anfängliche Schätzung der Posen der Sliding Window-Keyframes berechnet. Dieser Vorgang wird in Unterabschnitt 4.3.1 behandelt.

Dann werden im Anschluss diese Posen-Schätzungen entlang der metrisch korrekten IMU-Pre-Integration-Ergebnisse ausgerichtet. Das Verfahren dazu erläutert Unterabschnitt 4.3.2. Aus diesem Fusions-Vorgang werden Erkenntnisse über die Skalierung und Ausrichtung der Kamera-Keyframes entlang der Schwerkraft, die Geschwindigkeit und auch das Bias der IMU gewonnen.

#### 4.3.1. Vision-Only Structure from Motion in Sliding Window

In diesem Schritt der Initialisierungs-Prozedur wird eine erste Einschätzung der Kamerapositionen anhand visueller Informationen vorgenommen.

Auch hier verwendet der Algorithmus nur die Frames des Sliding Window, um die Komplexität der Berechnungen in Grenzen zu halten.

Zunächst wird untersucht, ob zwischen dem aktuellen und einem der letzten Keyframes des Sliding Window mehr als 30 Features wiedergefunden werden können. Außerdem muss der durchschnittliche rotations-kompensierte Parallaxen-Wert größer als 20 Pixel sein.

Ist das der Fall, handelt es sich um einen Keyframe. Mittels der Feature-Korrespondenzen der beiden Frames und der sogenannten „Fivepoint Method“ (Nister, 2004) kann die relative Rotation und Translation zwischen zwei Bildern bestimmt werden. Andernfalls wird der aktuelle Frame im Sliding Window beibehalten und auf neue Frames gewartet.

Da zu diesem Zeitpunkt noch keine metrische Bezugsgröße zur Verfügung steht, wird zunächst der Skalierungsfaktor auf einen temporären Wert festgelegt. Im nächsten Schritt werden dann die Features der beiden Frames trianguliert und so ihre räumliche Tiefe bestimmt. Durch Lösen des *Perspective-n-Point*-Problems wird im Anschluss die Kamerapose aller anderen Keyframes des Sliding Window eingeschätzt.

Zum Schluss wird noch ein Bundle Adjustment auf diese Posen angewendet, in dem der Reprojection-Error aller Features des Sliding Window minimiert wird. Genauere Informationen dazu folgen in Unterabschnitt 4.4.1.

Da zu diesem Zeitpunkt noch keine Positionierung im World-Frame bekannt ist, wird der erste Kamera-Frame als Referenz für alle Kamera-Posen und Feature-Positionen verwendet.

Aus den Kamera-Posen können mithilfe der bekannten extrinsischen Parameter (siehe Abschnitt 5.3) die zugehörigen IMU-Posen berechnet werden. Unbekannt ist dabei allerdings der Skalierungsfaktor. Dieser wird im nächsten Schritt durch die visuell-inertiale Ausrichtung ermittelt.

### 4.3.2. Visual-Inertial Alignment

Dieser Abschnitt erklärt die Schritte, die zur Ausrichtung der aus dem Vision-Only SfM-Algorithmus gewonnenen Kameraposen entlang der pre-integrierten IMU-Daten notwendig sind.

Low Cost IMU-Sensoren, die in Smartphones verbaut werden, haben häufig ein signifikantes Bias. Gerade was die Gyroskop-Messungen angeht, kann das bei der Ausrichtung einen negativen Einfluss auf die Genauigkeit der Schätzung haben. Daher wird vor der eigentlichen Ausrichtung zunächst das Gyroskop-Bias bestimmt und dann erst im zweiten Schritt Geschwindigkeit, Schwerkraft und der metrische Skalierungsfaktor berechnet.

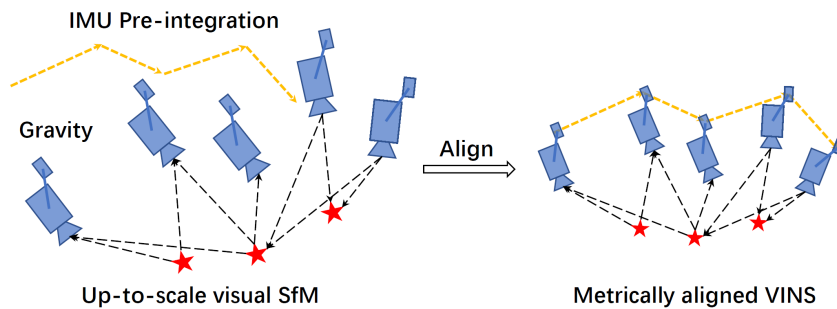


Abbildung 4.3.: Visual Inertial Alignment (Li et al., 2017, S. 5)

### 4.3.2.1. Gyroscope Bias

Betrachtet man zwei aufeinanderfolgende Frames des Sliding Window, kann die relative Rotation dieser beiden Frames aus dem Ergebnis der Vision-Only SfM entnommen werden. Außerdem ist die gleiche Rotation als Ergebnis der IMU-Pre-Integration bekannt. Die Genauigkeit letzterer ist aber von dem gesuchten Gyroskop-Bias abhängig.

Der Unterschied zwischen diesen beiden Rotationen ist ein quantifizierbarer Error. Mit diesem Error lässt sich in Abhängigkeit des Bias ein Least Square-Optimierungsproblem formulieren und für alle Frames des Sliding Window lösen.

Mit der neu gewonnenen Erkenntnis über das Gyroskop-Bias werden dann die ursprünglichen IMU-Pre-Integrations-Daten neu evaluiert.

### 4.3.2.2. Initialization of Velocity, Gravity and Metric Scale

Nachdem im vorherigen Schritt bildlich gesprochen die Krümmung der IMU-Daten korrigiert wurde, können nun im letzten Schritt der Initialisierungsphase die restlichen Unbekannten bestimmt werden. Diese umfassen die Geschwindigkeit, jeweils zu den Zeitpunkten der Keyframes, den Gravitationsvektor, der im World-Frame vertikal verläuft und den Skalierungsfaktor, der die visuellen SfM-Daten entlang der IMU-Pre-Integration-Daten ausrichtet. In Abbildung 4.3 ist dieser Vorgang abgebildet.

Auch dieses Ausrichtungs-Problem lässt sich wieder als Least Square-Problem formulieren und lösen.

Ist das geschehen, können die Positionen der visuellen Structure from Motion-Keyframes anhand des Faktors auf metrische Einheiten skaliert werden.

Fehlten bisher für die Bestimmung der Geschwindigkeit über die IMU-Messungen die Information über die initiale Geschwindigkeit, so wurden mit den Information der Keyframes und Features Fixpunkte in das System eingespeist, die Rückschlüsse auf diese Geschwindigkeit zulassen.

Nach dem Skalieren werden noch alle Variablen mithilfe des Gravitationsvektors von dem im SfM-Teil des Algorithmus temporär gesetzten Frame in den World-Frame rotiert.

### 4.3.3. Termination Criteria

Der Vision-Only SfM-Teil des Initialisierungsprozesses gilt als abgeschlossen, sobald der aufsummierte Reprojection-Error unter einen bestimmten Grenzwert fällt. Der Reprojection-Error ist ein Error-Maß, das durch den Abstand zwischen einem auf die Bildfläche projizierten 3D-Feature und dem in dem Bild beobachteten zugehörigen Feature definiert ist.

Die visuell inertielle Ausrichtung endet erfolgreich, wenn die Größe des extrahierten Gravitationsvektors der bekannten Schwerkraft von  $9.81 \text{ m/s}^2$  ähnlich genug ist.

Sobald diese beiden Bedingungen erfüllt sind, wird der Initialisierungsvorgang abgeschlossen und die metrischen Werte an die eng gekoppelte nicht lineare Optimierung weitergegeben.

#### 4.4. Tightly-Coupled Nonlinear Optimization

In diesem Abschnitt wird der Teil des Algorithmus beschrieben, der nach der Initialisierungsphase die Berechnung der Odometrie übernimmt. Dabei wird eine eng gekoppelte nicht lineare Optimierungsstrategie verwendet, um eine präzise und robuste Zustandseinschätzung möglich zu machen.

##### 4.4.1. Visual Inertial Bundle Adjustment Formulation

In diesem Unterabschnitt wird das Kernstück der nicht linearen Optimierungsstrategie, die Visual Inertial Bundle Adjustment Formulation vorgestellt:

$$\min_{\mathcal{X}} \left\{ \underbrace{\|\mathbf{r}_p - \mathbf{H}_p \mathcal{X}\|^2}_{\text{red}} + \underbrace{\sum_{k \in \mathcal{B}} \|r_{\mathcal{B}}(\hat{\mathbf{z}}_{b_k+1}^{b_k}, \mathcal{X})\|_{\mathbf{P}_{b_k+1}^{b_k}}^2}_{\text{green}} + \underbrace{\sum_{(l,j) \in \mathcal{C}} \|r_{\mathcal{C}}(\hat{\mathbf{z}}_l^{c_j}, \mathcal{X})\|_{\mathbf{P}_l^{c_j}}^2}_{\text{blue}} + \underbrace{\sum_{(l,i) \in \mathcal{L}} \|r_{\mathcal{L}}(\hat{\mathbf{z}}_l^{c_i}, \mathcal{X})\|_{\mathbf{P}_l^{c_i}}^2}_{\text{orange}} \right\} \quad (4.4)$$

(nach Li et al., 2017, S. 6)

Im Folgenden wird diese Formulierung erklärt:

Umklammert ist sie von der Minimum-Bedingung über dem Zustandsvektor  $\mathcal{X}$  des Sliding Window. Es handelt sich hier also wieder um ein Least Squares-Optimierungsproblem, das gelöst werden muss.

Der zu minimierende Term ist aus vier Blöcken zusammengesetzt, die in den nächsten Abschnitten genauer erklärt werden. Der rot umrandete Block stellt die Prior-Information dar. Unterabschnitt 4.4.5 erklärt woraus diese hervorgeht.

Das IMU Measurement Model (Unterabschnitt 4.4.2) ist grün umrandet, das Visual Measurement Model (Unterabschnitt 4.4.3) blau und das Loop Closure Model (Unterabschnitt 4.4.4) orange. Diese Modelle sind jeweils durch die aufsummierte Mahalanobis-Distanz des modellspezifischen Messungsrestwerts (Residual) definiert. Die Mahalanobis-Distanz berücksichtigt neben dem euklidischen Abstand auch die Wahrscheinlichkeitsverteilung durch Skalierung mittels der jeweiligen Kovarianzmatrix  $\mathbf{P}$ . Daher ist sie an dieser Stelle ein besseres Error-Maß.

Die Kovarianzmatrix beschreibt die Varianz eines mehrdimensionalen Zufallsvektors, der wiederum mit den spezifischen Komponenten des Residual zusammenhängt.

Ein Messungsrestwert (Residual) im Allgemeinen gibt den Unterschied zwischen einer beobachteten Messung und dem mit dem Zustandsvektor  $\mathcal{X}$  errechneten Zustand an. Wie die einzelnen Messungsrestwerte ermittelt werden, stellen die nächsten drei Unterabschnitte dar.

In der Implementation wird dieses nicht lineare Optimierungsproblem mit Hilfe der Ceres Solver-Bibliothek gelöst. Sie stellt für solche Zwecke Implementierungen der Gauss-Newton- und Levenberg-Marquardt-Methoden zur Verfügung.

#### 4.4.2. IMU Measurement Model

Das Residual des IMU Measurement Modells kann für die pre-integrierten Messungen, die zwischen  $b_k$  und  $b_k + 1$  liegen, wie folgt definiert werden:

$$r_{\mathcal{B}}(\hat{\mathbf{z}}_{b_k+1}^{b_k}, \mathcal{X}) = \begin{bmatrix} \delta \alpha_{b_k+1}^{b_k} \\ \delta \beta_{b_k+1}^{b_k} \\ \delta \gamma_{b_k+1}^{b_k} \\ \delta \mathbf{b}_{a_{b_k+1}}^{b_k} \\ \delta \mathbf{b}_{g_{b_k+1}}^{b_k} \end{bmatrix} \quad (4.5)$$

(Li et al., 2017, S. 6)

Mit den folgenden Bestandteilen:

- $\delta \alpha$  Delta der Position
- $\delta \beta$  Delta der Geschwindigkeit
- $\delta \gamma$  Delta der Orientierung
- $\delta \mathbf{b}_a$  Delta der Beschleunigungssensor-Bias
- $\delta \mathbf{b}_g$  Delta der Gyroskop-Bias

Die Berechnung anhand kinematischer Formeln ist recht komplex und wird der Einfachheit halber hier weggelassen. Für genauere Informationen sei auf Li et al. (2017) verwiesen. Wichtig ist, dass es sich dabei jeweils um die Differenz zwischen den Werten aus der IMU-Pre-Integration und berechneten theoretischen Werten handelt. Die Berechnung ist vom Zustandsvektor  $\mathcal{X}$  (siehe (4.1)) abhängig.

An dieser Stelle wird, anders als in der Initialisierungsphase, neben dem Gyroskop-Bias auch das Beschleunigungssensor-Bias beachtet und bei jeder nicht linearen Optimierung aktualisiert. Weichen die Ergebnisse zu stark von den zuvor geschätzten Werten ab, werden die Daten der IMU-Pre-Integration entsprechend neu evaluiert.

#### 4.4.3. Visual Measurement Model

Das Camera Measurement Residual des Visual Measurement Model wird definiert durch den Reprojection Error der Feature-Korrespondenzen. Dieser Error beschreibt die Differenz zwischen der gemessenen Feature-Bildkoordinate und der projizierten Koordinate des gleichen Features aus einem früheren Frame. Wie die Projektion durchgeführt und der Error berechnet wird, ist aus den folgenden zwei Gleichungen zu entnehmen. Dafür wird das  $l$ -te Feature in dem  $j$ -ten Kamerabild betrachtet.

Die erste Gleichung beschreibt die Reprojection des Features. Das Feature liegt in Form von normalisierten Bildkoordinaten  $u_l^{c_i}$  und  $v_l^{c_i}$  sowie der inversen Tiefe  $\lambda_l$  vor. Beide beziehen sich auf das erste Bild ( $i$ ), in dem dieses Feature beobachtet werden konnte.

$$\mathbf{f}_l^{c_j} = \begin{bmatrix} fx_l^{c_j} \\ fy_l^{c_j} \\ fz_l^{c_j} \end{bmatrix} = \mathbf{T}_c^{b^{-1}} \cdot \mathbf{T}_{b_j}^{w^{-1}} \cdot \mathbf{T}_{b_i}^w \cdot \mathbf{T}_c^b \cdot \frac{1}{\lambda_l} \cdot \begin{bmatrix} u_l^{c_i} \\ v_l^{c_i} \\ 1 \end{bmatrix} \quad (4.6)$$

(Li et al., 2017, S. 6)

Erklärung der Reprojection von rechts nach links:

- $\frac{1}{\lambda_l}$  Multiplikation mit Tiefe des  $l$ -ten Features
- $\mathbf{T}_c^b$  Transformation von Kameraframe zu Bodyframe
- $\mathbf{T}_{b_i}^w$  Transformation von Bodyframe zu Worldframe während des Frames  $i$

In den folgenden zwei Multiplikations-Schritten werden diese Transformationen invertiert und für Frame  $j$  angewandt. Als Resultat erhält man den Vektor  $\mathbf{f}_l^{c_j}$ , der die 3D-Position des  $l$ -ten Features im  $j$ -ten Bild angibt.

Das Residual ist dann mit den zuvor bestimmten Werten für  $fx_l^{c_j}$ ,  $fy_l^{c_j}$  und  $fz_l^{c_j}$  wie folgt definiert:

$$r_C(\hat{\mathbf{z}}_l^{c_j}, \mathcal{X}) = \begin{bmatrix} \frac{fx_l^{c_j}}{fz_l^{c_j}} - \hat{u}_l^{c_j} \\ \frac{fy_l^{c_j}}{fz_l^{c_j}} - \hat{v}_l^{c_j} \end{bmatrix} \quad (4.7)$$

(Li et al., 2017, S. 6)

Die beiden Differenzen sind dabei das Maß des Abstands zwischen dem projizierten Punkt und der tatsächlich in Frame  $c_j$  beobachteten Feature-Position. Auch hier wird wieder das normalisierte Pixelmodell eines Bildes mit den Koordinaten  $u$  und  $v$  verwendet. Zu beachten ist hierbei natürlich, dass auch diese Beobachtungen ein gewisses Rauschen aufweisen.

#### 4.4.4. Loop Closure Model

Dieses Residual fließt nur in die globale nicht lineare Optimierung ein, wenn ein Loop Closure erkannt wurde. Genauer zur Loop-Erkennung wird in Abschnitt 4.5 vorgestellt. Das Sliding Window wird dann an diesem Loop Closure-Keyframe verankert und mithilfe projizierter Feature-Korrespondenzen danach ausgerichtet.

Die Formulierung des Loop Closure Residual ist dem visuellen Messungsmodell sehr ähnlich:

$$\mathbf{f}_l^{c_m} = \begin{bmatrix} fx_l^{c_m} \\ fy_l^{c_m} \\ fz_l^{c_m} \end{bmatrix} = \mathbf{T}_{c_m}^{w^{-1}} \cdot \mathbf{T}_{b_i}^w \cdot \mathbf{T}_c^b \cdot \frac{1}{\lambda_l} \cdot \begin{bmatrix} u_l^{c_i} \\ v_l^{c_i} \\ 1 \end{bmatrix} \quad (4.8)$$

$$r_{\mathcal{L}}(\hat{\mathbf{z}}_l^{c_m}, \mathcal{X}) = \begin{bmatrix} \frac{fx_l^{c_m}}{fz_l^{c_m}} - \hat{u}_l^{c_m} \\ \frac{fy_l^{c_m}}{fz_l^{c_m}} - \hat{v}_l^{c_m} \end{bmatrix}$$

(Li et al., 2017, S. 7)

Betrachtet wird das Ganze für das  $l$ -te Feature im  $m$ -ten Loop Closure-Frame.  $\mathcal{L}$  sind die Feature-Korrespondenzen zwischen dem aktuellen und dem  $m$ -ten Frame der Keyframe-Datenbank.

Die Reprojection verläuft dabei genauso wie im Visual-Measurement-Modell, mit dem kleinen Unterschied, dass die Transformation  $\mathbf{T}_{c_m}^{w-1}$  schon die beiden inversen Transformationen Camera- zu Body-Frame und Body- zu World-Frame zusammenfasst. Sie wird festgelegt, wenn ein Keyframe das Sliding Window verlässt und in die Keyframe-Datenbank aufgenommen wird.

#### 4.4.5. Marginalization

Dieser Unterabschnitt erläutert das Vorgehen bei der Marginalisierung, dem Mechanismus zum Ausdünnen und Verschieben des Fensters.

Die Operationen zur Lösung der Optimierungsprobleme sind allgemein sehr rechenintensiv. Um die Prozessor-Belastung im Rahmen zu halten, werden in einem Marginalisierungsschritt ausgewählte Keyframe-Zustände und Features aus dem Sliding Window ausgegrenzt. Die Informationen dieser Zustände gehen aber nicht verloren, da sie als Prior (roter Block in Gleichung (4.4)) in die Posen-Optimierung mit einfließen.

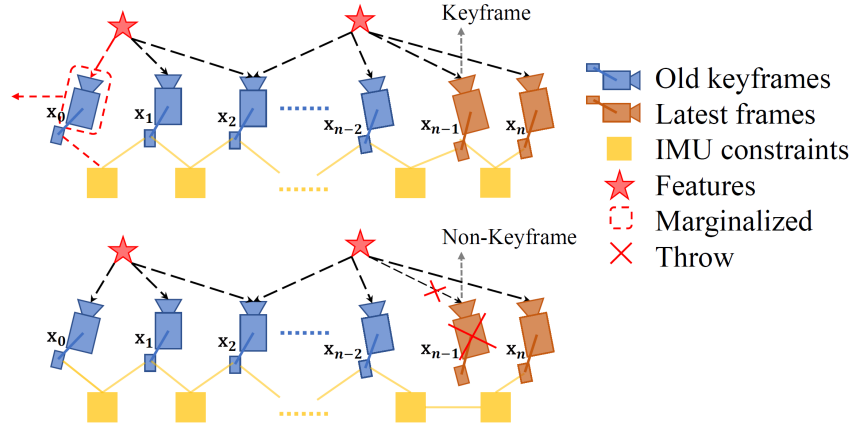


Abbildung 4.4.: Marginalization Strategy (Qin et al., 2017, S. 8)

Das verwendete Marginalisierungsverfahren unterscheidet zwischen zwei Fällen. Es betrachtet immer den vorletzten Frame. Handelt es dabei sich um einen Keyframe, wird der älteste Keyframe im Sliding Window zusammen mit seinen Bild- und IMU-Messungen in den Prior marginalisiert. Falls nicht, wird dieser Frame entfernt, wobei allerdings die IMU Messungen erhalten bleiben.

Eine Visualisierung dieses Vorgehens ist in Abbildung 4.4 zu sehen. Die obere Hälfte zeigt den Fall, dass der Keyframe-Status gegeben ist, und die untere, dass er fehlt. Diese Strategie wird angewandt, damit die Dichte des Systems gering bleibt und nur räumliche wichtige Keyframes erhalten bleiben.

## 4.5. Loop Closure

In diesem Abschnitt geht es um die Erweiterung der Visual Inertial Odometry um Funktionalitäten der Loop-Erkennung und Loop-Schließung.

Durch den Sliding Window-Ansatz fließt immer nur ein schmales Fenster der letzten Frames in die globale Optimierung ein. So kann im Verlauf der Zeit recht schnell ein signifikanter Posen-Drift auftreten. Um diesen Drift zu eliminieren und trotzdem die Echtzeitlauffähigkeit zu erhalten, wird parallel ein Algorithmus ausgeführt, der für die Loop-Erkennung und Schließung zuständig ist. Dieser Algorithmus erledigt dazu die folgenden Aufgaben:

1. Management der Keyframe-Datenbank (Unterabschnitt 4.5.1)
2. Finden von Kandidaten für eine Loop-Schließung (Unterabschnitt 4.5.2)
3. Relokalisierung bei Auftreten eines Loop (Unterabschnitt 4.5.3)

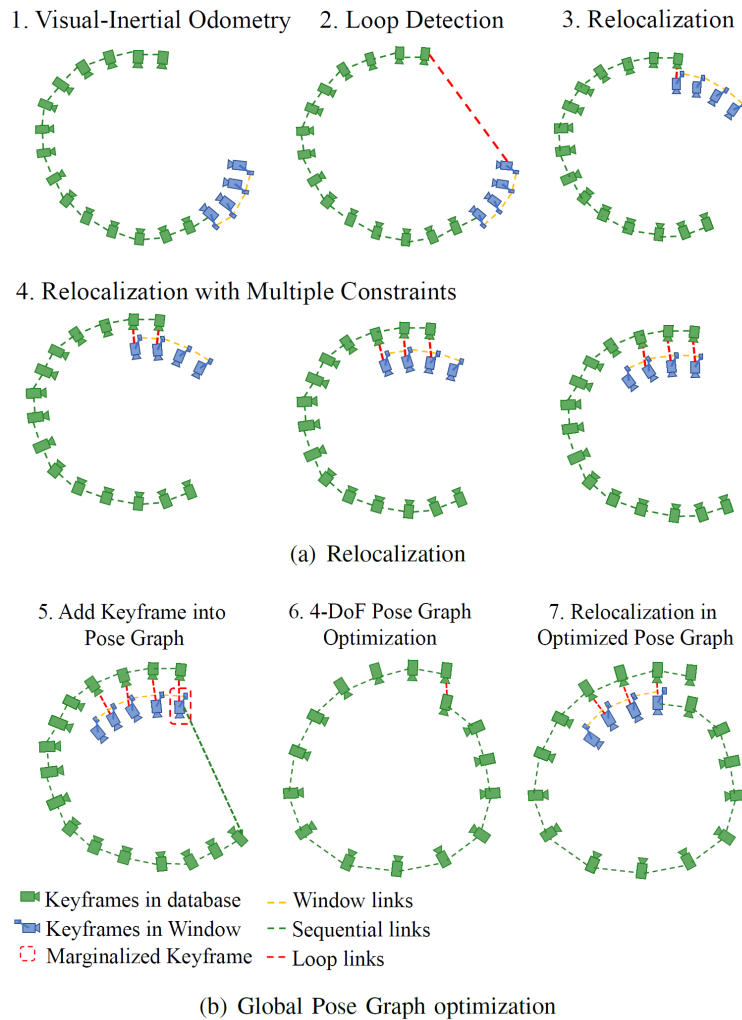


Abbildung 4.5.: Loop Closure (Qin et al., 2017, S. 10)



### 4.5.1. Keyframe Database

Die Keyframe-Datenbank ist die Speicherstruktur, in der die Keyframes gehalten werden, nachdem sie aus dem Sliding Window marginalisiert wurden.

In der Datenbank werden nur Informationen von vier Freiheitsgraden (DoF) gehalten, da nur die relative Position und Yaw-Rotation (um den Vektor der Schwerkraft) dem Drift unterliegen. Roll und Pitch sind durch die messbare Richtung der Schwerkraft davon nicht betroffen. Die Datenbank ist wie das Sliding Window in Form eines Graphen aufgebaut.

#### 4.5.1.1. Adding Keyframe into Pose Graph

In dem Posen-Graphen sind die einzelnen Keyframes durch zwei verschiedene Arten von Kanten verbunden. Neben den sequentiellen Kanten, die der zeitlichen Reihenfolge entsprechen, können nach einer Loop-Schließung Keyframes auch durch Loop-Closure-Kanten verbunden sein.

Keyframes werden immer genau dann in den Posen-Graphen aufgenommen, wenn sie das Ende des Sliding Window erreicht haben und marginalisiert wurden.

Die **sequentiellen Kanten** werden aus dem Sliding Window übernommen. Dabei werden nur die Translation und die Yaw-Rotation im Camera-Frame gespeichert.

**Loop-Closure-Kanten** werden immer nur zwischen einem Keyframe des Sliding Window und einem Keyframe der Datenbank geschlossen. Wenn der Keyframe am Ende des Fensters dann marginalisiert wird, wird auch für die Loop Closure-Kante nur die relative 4-DoF-Pose gespeichert. Außerdem wird eine globale Posen-Optimierung der Keyframe-Datenbank ausgelöst, wie in Abbildung 4.5(b) dargestellt ist.

#### 4.5.1.2. Pose Graph Management

Da im Laufe der Zeit die Datenbank stetig weiter anwächst, ist es in dem Echtzeitszenario der Anwendung wichtig, die Keyframes von Zeit zu Zeit auszudünnen. Nur so kann die Speicher- und Prozessorauslastung im Rahmen gehalten werden.

Kriterium für das Aussortieren von Keyframes ist, dass möglichst wenige Informationen verloren gehen. Daher werden Keyframes, die über eine Loop Closure-Kante mit einem anderen Keyframe verbunden sind, komplett davon ausgeschlossen. Andere Keyframes werden danach ausgewählt, ob sie sich signifikant in Orientierung und Position von ihren Nachbarn unterscheiden. Ist das nicht der Fall, können sie aussortiert und die beiden anliegenden sequentiellen Kanten zusammengefasst werden.

### 4.5.2. Loop Detection

Ziel der Loop-Erkennung ist es, schon einmal besuchte Orte wiederzuerkennen, um in späteren Schritten die aktuelle Position danach auszurichten.

Die Loop-Erkennung ist selber in zwei Schritte aufgeteilt. Im ersten Schritt sollen potenzielle Kandidaten gefunden und im zweiten Schritt Feature-Korrespondenzen zwischen dem besten Kandidaten und dem aktuellen Frame hergestellt werden.

Bei der Suche nach Kandidaten für ein Loop Closure werden immer der aktuelle Keyframe und alle Keyframes der Keyframe-Datenbank betrachtet. Um diese sehr komplexe Aufgabe zu lösen, wird ein „Bag-of-Words“ (BoW) Place Recognition-Algorithmus eingesetzt. Ursprünglich stammt dieser Ansatz aus dem Fachgebiet des *Natural Language Processing*.

Übertragen auf die Computer Vision entsprechen Features in den Bildern den *Words*. Das Vokabular setzt sich aus allen beobachteten Features zusammen. Die Bilder der Kamera werden nicht gespeichert, um die Speicheranforderungen gering zu halten.

Ein Bild kann dann als *Bag-of-Words*, ein Histogramm dieses Vokabulars, dargestellt werden, wobei räumliche Zusammenhänge erst einmal unbeachtet bleiben. Räumlich nahe Bilder haben aber zwangsläufig eine ähnliche Histogramm-Darstellung, anhand welcher Paare gefunden werden können.

Die verwendete Implementation ist DBow2 (Gálvez-López and Tardós, 2012) mit dem BRIEF-Feature-Descriptor (Calonder et al., 2010) als Feature-Repräsentationsform.

Damit der beste gefundene Loop-Closure-Kandidat für die Relokalisierung verwendet werden kann, müssen zunächst noch Feature-Korrespondenzen gefunden werden. Diese können nicht direkt aus dem Bag-of-Words entnommen werden.

Dabei werden zwei verschiedene Strategien verwendet:

1. Bei kleinem Drift genügt die Reprojection der 3D-Features des aktuellen Frames in den Loop Closure-Kandidaten. Auf das Ergebnis der Reprojection wird anschließend der KLT Feature Tracker angewendet, um Korrespondenzen zu finden.
2. Bei größerem Drift kann der Tracker keine Korrespondenzen ermitteln, da die Pixelverschiebung der Features zu groß ist. Für diesen Fall werden mithilfe des BoW-Ansatzes und kleineren Feature-Paketen Histogramm-Paare aus der direkteren Nachbarschaft von beobachteten Features ermittelt. Per Subsampling mit RANSAC werden Outlier aus den erhaltenen Paaren ausgesiebt und das beste Match als Feature-Korrespondenz behandelt.

##### 4.5.3. Relocalization

Ist ein Loop Closure erkannt, so wird dieser in Form der Feature-Korrespondenzen in das Loop Closure Model (Unterabschnitt 4.4.4) und damit in die nicht lineare Optimierung des Sliding Window eingefügt. Er fungiert dadurch als neuer Anker des Sliding Window. Durch diese Art der Einbindung ist es auch möglich mehrere Loop Closure gleichzeitig zu beachten und zu verarbeiten. Dadurch wird die Genauigkeit der Relokalisierung noch verbessert. Dies ist in Abbildung 4.5(a) 4. abgebildet.

Wie bereits in Unterabschnitt 4.5.1 angeschnitten, gibt es auch für die Keyframe-Datenbank einen Optimierungsprozess ihrer Keyframes. Dieser Prozess ist in Abbildung 4.5(b) zu sehen. Ausgelöst wird die Optimierung, wenn ein Keyframe des Sliding Window marginalisiert wird und eine Loop Closure-Kante zwischen ihm und einem Keyframe der Datenbank besteht.

## 4.6. Eignung für mobile Geräte

In diesem Abschnitt werden die wichtigsten Merkmale und Anpassungen dieses Algorithmus erklärt, die dafür sorgen, dass er in Echtzeit auf modernen Smartphones laufen kann.

Im Gegensatz zu anderen optimierungsbasierten Algorithmen, die den gesamten globalen Posen-Graphen in die Optimierung mit einschließen, bewirkt der Sliding Window-Ansatz eine signifikante Einsparung an Prozessorleistung. Außerdem lässt er die Komplexität über den Verlauf der Zeit nicht stetig größer werden. Auch die Pre-Integration hat einen positiven Effekt auf die Komplexität des Optimierungsproblems.

Ursprünglich wurde der Algorithmus für den Einsatz in ROS, dem Robot Operating System konzipiert und implementiert. Die Portierung auf iOS beinhaltet einige Optimierungen, die die Echtzeitlauffähigkeit auf mobilen Geräten erst ermöglichen. Diese Optimierungen und Kompromisse werden im Folgenden erklärt:

Zunächst einmal ist der Kamera-Input auf eine Frequenz von 30 Hz und eine Auflösung von  $640\text{px} \times 480\text{px}$  beschränkt. Der IMU-Input kann durch die Pre-Integration uneingeschränkt mit der Frequenz von 100 Hz genutzt werden.

Außerdem wird die Ausführung der verschiedenen Bestandteile des Algorithmus auf verschiedene Threads aufgeteilt, sofern sie parallel ablaufen können. Diese Threads sind wie folgt aufgeteilt:

1. Feature Detection - 10 Hz mit einem Maximum von 60 neuen Features  
Feature Tracking - 30 Hz
2. Initialisation / nonlinear Optimization - 10 Hz  
mit der Sliding Window-Größe von 10 Keyframes
3. Loop Detection and Feature Retrieval - 3 Hz  
Relocalisation, wenn mehr als 10 Feature-Korrespondenzen gefunden werden

## 5. Portierung

In diesem Kapitel werden das Vorgehen und die Hindernisse bei der Portierung des iOS Projektes auf die Android-Plattform vorgestellt. Gegliedert ist das Kapitel in die Einbindung der externen Bibliotheken (Abschnitt 5.1), die Details zu der Übersetzung des iOS spezifischen Codes (Abschnitt 5.2) und das Vorgehen, das gewählt wurde, um die verschiedenen benötigten gerätespezifischen Parameter zu ermitteln (Abschnitt 5.3).

### 5.1. Externe Bibliotheken

VINS Mobile ist von mehreren verschiedenen externen Bibliotheken abhängig, die in dem iOS-Projekt größtenteils für iOS vorkompiliert sind.

In diesem Abschnitt werden die verwendeten Software-Bibliotheken vorgestellt und es wird der Vorgang beschrieben, der nötig war, um sie in das Projekt einzubinden und auf der Android-Plattform nutzen zu können. Zu diesen Bibliotheken gehören Boost, Eigen, OpenCV, Ceres Solver, DBoW2 und die GNU Standard Template Library.

CMake ist inzwischen das Standard Build System für das Android Native Development Kit (NDK). ndk-build ist eine andere und ältere Möglichkeit, C und C++ Code für Android zu kompilieren.

Da XCode, die Entwicklungsumgebung für iOS, jedoch ein anderes Build System benutzt, musste die Struktur des CMake Dateien-Systems von Grund auf komplett neu aufgebaut werden. Dazu wurden zunächst für die verschiedenen Unterordner jeweils eine CMake-Datei angelegt, um zu verhindern, dass der gesamte Kompilervorgang in einer einzigen unübersichtlichen großen Datei beschrieben wird. Da die einzelnen Unterordner untereinander Abhängigkeiten besitzen, wurden diese Abhängigkeiten im Code analysiert und die Erkenntnisse in die CMake-Dateien in Form von Link-Befehlen eingebunden.

Dabei ist die in Abbildung 5.1 dargestellte hierarchische Struktur entstanden. In den links angegebenen Pfaden liegen `CMakeLists.txt`-Dateien, die in CMake mit dem Befehl `add_subdirectory` verknüpft werden können. In Anführungszeichen stehen die Bibliotheken, die über den `add_library`-Befehl entweder importiert oder aus Source-Dateien gebaut werden.



Abbildung 5.1.: CMake Hierarchie

### 5.1.1. Boost

Boost ist eine C++ Library, die viele verschiedene nützliche Grundfunktionen bietet. Einige Teile wurden im Verlauf der Zeit in die Sprache selbst integriert.

In dem VINS Projekt wurde nur die Klasse `dynamic_bitset` aus dieser Bibliothek verwendet. Zum Einsatz kommt sie im Zusammenhang mit der Loop-Detection bzw. dem Bag-of-Words-Algorithmus, um einfachen Zugriff auf einzelne Bits der Feature-Deskriptoren zu bekommen.

#### Einbindung

Die für die iOS-Prozessorarchitektur vorkompilierte statische Bibliothek ist auf Android nicht nutzbar.

Über eine in die CMake-Android-Toolchain integrierte Methode kann Boost als Bibliothek mit dem einfachen Befehl `find_package(Boost ...)` eingebunden werden. Die dabei verwendete Version von Boost ist allerdings von der CMake-Version abhängig.

Die von VINS Mobile verwendete Boost-Version ist 1.63, welche erst von CMake 3.7 oder höher verwendet wird. Die aktuelle in der Android Studio Toolchain verwendete CMake-Version ist allerdings 3.6.4.

Die CMake-Version der Toolchain manuell auf 3.7 zu aktualisieren, war nicht ohne Probleme möglich. Umgangen wurde das Problem, indem die Bibliothek separat in der richtigen Version heruntergeladen und als `INTERFACE`-Bibliothek ausschließlich über die Header-Dateien eingebunden wurde. Das ist nicht für alle Module eine Option, aber da sich die benötigte Klasse nicht unter den Modulen befindet, die vorkompiliert werden müssen, ist es an dieser Stelle möglich.

### 5.1.2. Eigen

Eigen<sup>1</sup> ist eine C++ Template Library für lineare Algebra. Sie umfasst Datenstrukturen wie Matrizen und Vektoren und bietet daneben auch verschiedenen Algorithmen, die auf diesen operieren.

Außerdem bietet sie neben den Vorteilen, die Templates in C++ durch die Typisierung von Template-Parametern zur Kompilierzeit mit sich bringen, Optimierungsmöglichkeiten für spezielle Prozessorbestandteile, die „Same Instruction Multiple Data“ (SIMD)-Verarbeitung implementieren. Bekannte Befehlssätze dafür sind z. B. SSE von Intel oder eben NEON für Android.

#### Einbindung

Eigen ist eine Template Library, die komplett aus Header-Dateien besteht. Es muss deswegen nichts kompiliert werden, was das Einbinden sehr einfach macht. Dazu muss nur das *include directory*, in dem die Header-Dateien liegen, CMake bekannt gemacht werden und die Bibliothek mit dem Parameter `INTERFACE` hinzugefügt werden.

Verwendet wird Eigen außer in den Projekt-Source-Dateien auch von der Ceres Solver-Bibliothek, die in Abschnitt 5.1.4 vorgestellt wird.

### 5.1.3. OpenCV

OpenCV ist eine weitverbreitete Open Source-Bildverarbeitungsbibliothek. Sie bietet sehr viele verschiedene Funktionalitäten für Anwendungen im Bereich der Computer Vision. Außer der in diesem Projekt verwendeten C++ Version werden auch Schnittstellen für C, Python und Java angeboten.

In dieser Anwendung wird sie hauptsächlich für das Erkennen und das Tracking der Bild-Features in aufeinanderfolgenden Kamerabildern verwendet. Einen weiteren Einsatzzweck hat sie im Konvertieren und Rendern des Ausgabebildes.

#### Einbindung

Für das iOS-Projekt wird ein Paket aus Header und vorkompilierten statischen Bibliotheksdateien zum Download zur Verfügung gestellt. In den Projekt-Meta-Dateien selbst wird diese Version als „A weird customized version based on 3.0.0“ betitelt.

Wie auch schon bei der Boost Library in Abschnitt 5.1.1 kann die vorkompilierte Version nicht auf Android verwendet werden. Anders als bei Boost kann OpenCV allerdings nicht auf die gleiche Art ausschließlich über die Header-Dateien eingebunden werden. Daher musste in diesem Fall das OpenCV Android SDK Package heruntergeladen und eingebunden werden. Es wurde dafür die zu dem Zeitpunkt neueste Version 3.4.0 ausgewählt. Der Download enthält neben einigen Beispielen sowohl die Java-Schnittstelle, als auch die für die verschiedenen Android-Prozessorarchitekturen vorkompilierte native C++ Bibliothek, die verwendet wurde.

Nachdem in den entsprechenden CMake-Dateien der Dateipfad der Bibliothek bekannt gemacht wurde, kann die Bibliothek einfach über den CMake-Befehl `find_package` importiert werden.

---

<sup>1</sup><http://eigen.tuxfamily.org/>

### 5.1.4. Ceres Solver

Ceres Solver (Agarwal et al., [n. d.]) ist eine sehr umfangreiche Open Source C++ Bibliothek von Google zum Modellieren und Lösen komplizierter nicht linearer Optimierungsprobleme. In der Implementierung des Algorithmus findet sie starke Verwendung unter anderem in der initialen Structur from Motion-Prozedur und in der eng gekoppelten Optimierung des Posen-Graphen im Sliding Window. Sie ist somit zentraler Bestandteil des Framework.

Die Integration auf Android ist aber alles andere als leicht umzusetzen gewesen.

#### Einbindung

Wie auch schon für Boost und OpenCV wurde Ceres für iOS vorkompiliert in dem Projekt mitgeliefert. Da es aber anders als bei Boost nicht möglich war, nur die Header-Dateien zu nutzen und die Bibliothek auch nicht für Android vorkompiliert angeboten wird, musste sie, um sie nutzen zu können, selbst kompiliert werden.

Dieser Vorgang hat sich als sehr zeitraubend erwiesen, da er nicht besonders gut dokumentiert ist und häufig Fehler auftraten.

Für alle Plattformen außer Android nutzt Ceres das CMAKE Build-System. Nur für Android wird das ältere und in diesem Projekt eigentlich nicht genutzte ndk-build verwendet. Da die Einbindung dieses Systems in den CMake Build-Prozess nicht ohne weiteres möglich ist, wurde die Build-Prozedur manuell in der Kommandozeile ausgeführt. Zu finden ist die ausführbare `ndk-build`-Datei in dem Android-SDK-Installationsverzeichnis in dem Ordner `ndk-bundle`.

Zur Konfigurierung der Compiler-Einstellungen verwendet dieses System die `Application.mk`- und `Android.mk`-Dateien, die in dem Verzeichnis liegen, in dem es aufgerufen wird.

Mit den voreingestellten Parametern sind mehrere Setup-Probleme aufgetreten, die weitere Anpassungen der Compiler-Parameter erforderten.

Ceres hängt selbst von einigen weiteren Bibliotheken ab. Viele davon sind aber optional, sodass nur Eigen als externe Bibliothek eingebunden werden musste. Dazu musste wie auch schon bei CMake lediglich der Include-Pfad definiert werden.

Die in der `Application.mk`-Datei standardmäßig ausgewählte Implementierung der Standard C++ Bibliothek ist `libc++` des LLVM<sup>2</sup>-Projekts. Eigentlich ist diese Bibliothek auch die Standardeinstellung eines neuen Android Studio-Projekts. Trotzdem kam es bei dem Link-Vorgang zu Problemen. Da unterschiedliche Arten des so genannten „name mangling“, dem Erzeugen eines eindeutigen Strings, genannt Symbol, aus der Methodensignatur des Quellcodes, verwendet wurden, konnte der Linker die im Programm gesuchten Methoden der Bibliothek nicht finden.

Zur Analyse der Symbole der kompilierten und zu einer statischen Bibliotheksdatei gepackten C++ Quelldateien, kann das in dem Android NDK inbegriffene Programm `nm.exe` verwendet werden. Zu finden ist es im NDK Installationsverzeichnis unter dem Pfad `ndk-bundle\toolchains\arm-linux-androideabi-4.9\prebuilt\windows-x86_64\arm-linux-androideabi\bin`.

---

<sup>2</sup><http://www.llvm.org/>

Gelöst wurde das Problem, indem in Android Studio über den Parameter `cppFlags` CMake explizit mitgeteilt wurde, dass die eben genannte Implementation der Standard Bibliothek verwendet werden soll. Später musste diese Einstellung noch einmal geändert werden, siehe dazu Unterabschnitt 5.1.6.

Dank der großen Anzahl verschiedener Hersteller existieren unter Android mehrere Prozessorarchitekturen. Die aktuell bei einem neuen NDK-Projekt standardmäßig eingestellten Befehlssätze sind `arm64-v8a`, `armeabi-v7a`, `x86` und `x86_64`. In den Build-Optionen der `Application.mk` ist ausschließlich `armeabi-v7a` aktiviert. Das Testgerät unterstützt `arm64-v8a`. Um alle Vorteile dieses 64-Bit-Befehlssatzes nutzen zu können, wurde die Option entsprechend angepasst.

Damit die Kompatibilität zwischen der im Android Studio Projekt gewählten minimalen Android SDK Version 24 und der kompilierten Ceres Bibliothek gewährleistet ist, musste der Parameter `APP_PLATFORM` auf `android-24` gesetzt werden.

Die von Ceres empfohlene Logging-Bibliothek *google-glog* lässt sich in der aktuellen Version nicht für Android kompilieren. Um Ceres trotzdem benutzen zu können, liegt der Bibliothek die minimale Ersatz-Bibliothek *minilog* bei. Sie besitzt Nachteile in Performance und Funktionsumfang. Damit minilog von dem Compiler verwendet werden kann, ist das manuelle Kopieren ihrer Header-Dateien in den dafür vorgesehenen Ordner notwendig.

Die `config.h`-Header-Datei wird, anders als bei dem CMake Build-Vorgang der anderen von Ceres unterstützten Systeme, über `ndk-build` nicht automatisch erstellt und auch nicht den Compiler-Optionen entsprechend konfiguriert. Daher musste ich mir die Entsprechung zwischen den Optionen der `Android.mk`-Datei und den defines der `config.h`-Datei selbst erschließen.

### 5.1.5. DBoW2

DBoW2 (Gálvez-López and Tardós, 2012) ist eine Open Source C++ Bibliothek, die verwendet werden kann, um Bilder mittels Feature-Deskriptoren in eine Bag-of-Words-Vektor-Repräsentation umzuwandeln. Außerdem bietet sie die Funktionalität einer Datenbank, mit welcher im Anschluss effizient nach Übereinstimmungen mit früheren Bildern gesucht werden kann.

#### Einbindung

Die Dateien dieser Bibliothek sind eng verknüpft mit den Source-Dateien des VINS-Frameworks. Da aber sowohl die Header- als auch die Source-Dateien in entsprechenden Ordnern in dem Projekt mitgeliefert werden, war es kein Problem, sie zusammen zu kompilieren. DBoW2 hat selbst noch zwei Dependencies, die beachtet werden müssen: zum einen OpenCV, welches ja auch an anderer Stelle in diesem Projekt benötigt wird, und zum anderen DLib<sup>3</sup>. Die Source-Dateien dieser Bibliothek waren aber auch neben den DBoW-Dateien vorhanden.

---

<sup>3</sup><https://github.com/dorian3d/DLib>



### 5.1.6. GNU C++ Library

Unter Android stehen verschiedene C++ Runtime Libraries zur Auswahl, unter anderem `gnu_stl`. Die in der aktuellsten Android Studio-Version standardmäßig verwendete Version ist eigentlich die LLVM `libc++`. Bei dem Kompilieren mit dieser STL treten aber Probleme mit OpenCV auf, sodass sie nicht zu verwenden ist. Deswegen musste hier die GNU C++ Standard Template Library genutzt werden.

#### Einbindung

Um Kompatibilitätsprobleme zu vermeiden, wurde neben dem CMake Build-Vorgang der eigentlichen Applikation in Android Studio auch der `ndk-build` Build der Ceres Library entsprechend angepasst und neu kompiliert. Zwischen den beiden Buildsystem (CMake und `ndk-build`) bestehen kleinere syntaktische Unterschiede.

Unter CMake muss diese Einstellung in der Datei `build.gradle` im Verzeichnis `app` mit dem CMake-Argument `-DANDROID_STL=gnustl_static` vorgenommen werden. Bei `ndk-build` ist das in der entsprechenden `Application.mk`-Datei im Verzeichnis der Ceres-Bibliothek durch das Setzen der Option `APP_STL:=gnustl_static` möglich.

## 5.2. Übersetzung

Obwohl der Großteil des Frameworks als C++ Code vorliegt, gab es doch einen erheblichen Teil, der iOS-spezifischen Code enthielt. Dieser Code hat vor allem die Aufgabe, die Sensor-Inputs der Kamera und des Bewegungssensors so zu verarbeiten, dass sie in den eigentlichen Algorithmus eingespeist werden können. Außerdem managt er die Inputs des Users und den Output auf dem Bildschirm. Die verwendete Programmiersprache war Objective C.

In diesem Abschnitt werden die Herangehensweise an die Übersetzung und die Auffälligkeiten und Probleme beleuchtet, die dabei auftraten.

Unterabschnitt 5.2.1 zeigt die Herangehensweise in der zeitlichen Abfolge auf, Unterabschnitt 5.2.2 stellt die iOS Projektstruktur vor, der Aufbau der Android-Applikation wird in Unterabschnitt 5.2.3 erklärt und im Anschluss wird in Unterabschnitt 5.2.4 auf die Gemeinsamkeiten und Unterschiede zwischen iOS und Android eingegangen. Zuletzt werden in Unterabschnitt 5.2.5 noch die Probleme erklärt, die bei der Portierung des Frameworks aufgefallen sind.

### 5.2.1. Herangehensweise

In diesem Abschnitt wird das Vorgehen, das zur Portierung des iOS spezifischen Codes gewählt wurde, zeitlich chronologisch betrachtet.

Wie bei den externen Bibliotheken wurde auch hier zunächst einmal analysiert, wie das Projekt aufgebaut ist. Da dies mein erster Kontakt mit der Entwicklung auf iOS mit Objective C war, war die Orientierung im Code nicht so einfach wie unter Android und C++ möglich. Erschwerend kam hinzu, dass die Portierung auf Windows durchgeführt

wurde und daher XCode nicht zum Experimentieren zur Verfügung stand. Es musste deswegen häufig die Apple Developer Documentation<sup>4</sup> verwendet werden.

Eine Übersicht über die Objective C-Programmteile wird in Unterabschnitt 5.2.2 gegeben.

Nachdem im Groben der Programmkontrollfluss und die Funktionen der einzelnen Bestandteile bekannt waren, konnte die Übersetzung beginnen. Es wurde versucht, so viel Code wie möglich im Originalzustand zu belassen und nur die nötigsten iOS spezifischen Funktionen zu übersetzen. Dazu wurde zu den jeweiligen iOS-Funktionen, wie beispielsweise der Initialisierung der Kamera, das entsprechende Pendant für Android gesucht.

Da es auf Android nie vollständig möglich ist, eine reine C++ Anwendung zu programmieren, und teilweise die Verwendung von Java in performance-unkritischen Teilen mehr Sinn ergibt, sind einige Teile in Java implementiert und andere wiederum in C++. Die genaue Struktur, die dabei entstanden ist, wird in Unterabschnitt 5.2.3 beleuchtet.

Bei der Übersetzung und dem stetigen Testen dieser sind einige Probleme ans Licht gekommen, die mit der Android-Plattform und -Hardware zusammenhängen. Genauere Details dazu und zu deren Lösung sind in Unterabschnitt 5.2.5 beschrieben.

### 5.2.2. Objective C Dateien

In der Analyse des iOS Projekts wurden die Objective C-Dateien und -Klassen auf ihre Funktionen und Verbindungen untersucht. In den folgenden Unterabschnitten wird ein Überblick über diese Dateien gegeben.

#### **main**

Die **main**-Datei enthält die **Main**-Methode und ist der Einstiegspunkt der Anwendung. Hierin wird das User Interface angelegt und diesem die AppDelegate-Klasse bekannt gemacht. Damit wird der Event Cycle gestartet. Die Events werden von der AppDelegate-Klasse verarbeitet.

#### **AppDelegate**

In der Klasse AppDelegate wird das Application State Management definiert, indem festgelegt wird, wie auf die verschiedenen Lifecycle-Events reagiert werden soll. Dazu gehören unter anderem die Fragen, was passieren soll, wenn die App geschlossen wird, pausiert oder wieder geöffnet wird. Alle Events wurden bei der automatisch generierten Standard-Implementierung belassen und kein zusätzlicher Code hinzugefügt.

#### **CameraUtils**

Die Hilfsklasse **CameraUtils** hat nur eine einzige Funktion: Sie legt das **ExposureTargetBias** fest. Die Einheit ist der so genannte „exposure value“ (EV), der durch die Kombination der Belichtungsdauer und der Blendenzahl abgeleitet wird.

---

<sup>4</sup><https://developer.apple.com/documentation>

Damit hat diese Einstellung einen Effekt auf die Beleuchtungsdauer des Bildes, ohne die *auto exposure* zu deaktivieren.

Durch das Setzen eines negativen Offsets soll in diesem Anwendungsfall ein dunkleres Bild im Tausch gegen eine geringere Belichtungszeit in Kauf genommen werden. Diese ist wichtig, um die getrackten Features nicht durch Auftreten von starker Bewegungsunschärfe zu verlieren. Ein anderer Grund ist das Abschwächen der Rolling-Shutter-Artefakte, die bei fast allen Smartphone-Kameras auftreten und den Structur from Motion-Algorithmus stören.

Außerdem wird noch der Effekt der vorgenommenen Einstellung auf die Belichtungszeit ausgegeben.

### PJRSignatureView

Die Klasse `PJRSignatureView` ist eine Unterklasse der iOS UIKit-Klasse `UIView`, die die Möglichkeit bietet, aus Punkten mittels Bézierkurven eine weiche Linie zu erzeugen, um eine Unterschrift einzugeben und abzubilden.

In diesem Projekt wird diese Klasse jedoch nicht verwendet, daher muss davon ausgegangen werden, dass sie ein Überbleibsel aus der Entwicklung ist. Eventuell wurde sie für die Visualisierung des Bewegungsverlaufs in Betracht gezogen.

### VINS\_ios-Prefix

In dieser Datei werden einige Imports definiert, unter anderem OpenCV und das dem iOS SDK angehörige UIKit.

Außerdem verursacht diese Datei eine Warnung, sollte das Projekt für eine SDK-Version geringer als 5.0 kompiliert werden.

### ViewController

Die Klasse `ViewController` kommt durch Einbindung in der Storyboard-Datei `Main.storyboard` zur Ausführung. Storyboard-Dateien sind iOS-spezifische XML-Dateien, die das User Interface und Übergänge zwischen Szenen beschreiben.

In dieser Klasse wird der Hauptteil des Algorithmus ausgeführt. Sie managet die Daten-Objekte und den Ablauf, verbindet die Inputs der Sensoren und des Benutzers mit dem Algorithmus und gibt den Output des Algorithmus über das Display aus.

Aufgeteilt ist diese Klasse in drei Dateien mit den Endungen `.h`, `.m` und `.mm`. Die `.mm`-Endung bedeutet, dass es sich hierbei um eine Objective-C++ Datei handelt. In einer solchen Datei ist es auch möglich, C++ Funktionalitäten zu verwenden. Der größte Teil des Codes und der Funktionalität befindet sich in eben dieser `ViewController.mm`-Datei.

Die wichtigsten Methoden sind im Folgenden kurz zusammengefasst:

*viewDidLoad*

Wird aufgerufen, sobald das zugehörige UI-Element, die View, geladen ist. Führt verschiedene Initialisierungen und Einstellungen durch und startet die Kamera und die verschiedenen Threads.

<i>processImage</i>	Verarbeitet den Kamera Input Frame durch Feature-Erkennung und Tracking und rendert das Ausgabebild je nach Einstellung und Status entweder mit Augmented Reality Overlay oder mit dem Bewegungsverlauf.
<i>process</i>	In separatem Thread: Führt je nach aktuellem Status entweder die Initialisierung oder die eng gekoppelte nicht lineare Optimierung aus.
<i>loop_thread</i>	In separatem Thread: Analysiert, ob es zwischen dem aktuellen Frame und der Keyframe-Datenbank zu einem Loop gekommen ist.
<i>globalLoopThread</i>	In separatem Thread: Führt die Optimierung des Posen-Graphen der Keyframe-Datenbank durch.
<i>imuStartUpdate</i>	Initialisiert die Listener für Beschleunigungssensor und Gyroskop.

Daneben gibt es noch einige weitere Methoden für User-Interaktion und Nutzung des Speichersystems.

### **ViewControllerPublic**

Hierbei handelt es sich um eine leere Header-Datei, die in dem Projekt nicht verwendet wird.

### **5.2.3. Android-Applikationsaufbau**

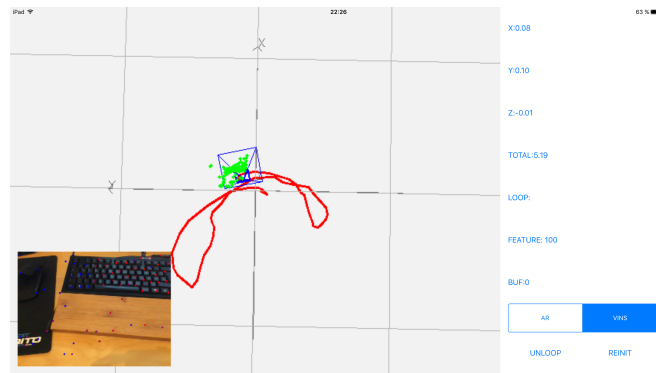
In diesem Unterabschnitt wird der Aufbau der Android-Applikation erklärt. Es wird beschrieben, welcher Teil welche Aufgaben übernimmt und wieso diese Aufteilung gewählt wurde.

Im Groben lässt sich die Struktur in drei Teile aufteilen: den Java- und UI-Teil, der unter 5.2.3.1 behandelt wird, den C++ Teil, der unter 5.2.3.3 zu finden ist und den JNI-Teil, der die beiden verbindet und unter 5.2.3.2 beschrieben wird.

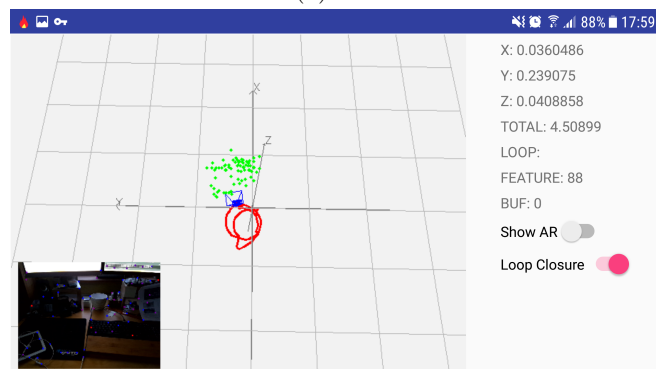
Einige Funktionen wurden aus zeitlichen Gründen nicht umgesetzt, da sie für die Funktionalität der Anwendung nicht von großer Bedeutung waren. Dazu gehören alle Methoden und Schnittstellen, die der Aufnahme und Offline-Speicherung der Input-Daten dienen, sowie die, die diese Daten laden und auswerten. Diese Methoden werden in dem Modus verwendet, der die Inputs nicht in Echtzeit verarbeitet.

In Abbildung 5.2 kann die dem iOS User-Interface nachempfundene Android-Benutzeroberfläche im Vergleich betrachtet werden.

## 5. Portierung



(a) iOS



(b) Android

Abbildung 5.2.: Vergleich der User Interfaces

### 5.2.3.1. Java und UI

Dieser Teil umfasst die `MainActivity.java` und die dazugehörige `activity_main.xml`-Layoutdatei. Letztere definiert, ähnlich wie die storyboard-Dateien unter iOS, das UI-Layout mit der Anordnung der UI-Komponenten.

Zu diesen Komponenten zählen:

- eine **TextureView** für die Ausgabe des Kamerabildes mit AR-Overlay oder des Bewegungsverlaufs,
- eine **ImageView** für das Bild mit den Instruktionen für die Initialisierung,
- mehrere **TextViews** für die Ausgabe von Werten und Fehlermeldungen und
- zwei **Switches** für das AR-Overlay und für die Loop Closure-Funktionalität.

In Abbildung 5.2 ist das beschriebene User Interface dargestellt.

## 5. Portierung

In der Datei `MainActivity.java` werden folgende Funktionen implementiert:

<i>initLooper</i>	Startet einen separaten Thread, der den Kamerainput verarbeitet.
<i>initVINS</i>	Erzeugt ein neues Objekt des VINS Java Native Interface und ruft die Initialisierungsmethode auf.
<i>initViews</i>	Holt Zugriff auf alle UI-Elemente, um sie später im Code manipulieren zu können. Außerdem wird der <code>SurfaceTextureListener</code> der <code>TextureView</code> auf die <code>this</code> -Referenz gesetzt, damit die Callback-Funktion <code>onSurfaceTextureAvailable</code> ausgeführt wird, sobald die <code>TextureView</code> bereit ist.
<i>onSurfaceTextureAvailable</i>	In dieser Funktion werden die Kamerakonfiguration und die Erlaubnis überprüft, ob die Kamera verwendet werden darf. Am Ende wird die Kamera in dem separaten Thread gestartet und ein <code>CameraDevice.StateCallback</code> übergeben.
<i>cameraDeviceStateCallback</i>	Diese Methode wird in der Callback-Methode <code>onOpened</code> des <code>CameraDevice.StateCallback</code> -Objekts ausgeführt. Hier werden weitere Kameraeinstellungen vorgenommen, wie z. B. das Bildformat, das Auto Exposure Offset (auf Android Compensation genannt) und die Bildwiederholungsrate der Kamera. Weiterhin werden ein <code>ImageReader</code> instanziiert, der für den Zugriff auf die einzelnen Bilder zuständig ist und eine <code>CaptureSession</code> erzeugt, der wiederum ein <code>CameraCaptureSession.StateCallback</code> übergeben wird.
<i>updateCameraView</i>	Diese Methode wird in der Callback-Methode <code>onConfigured</code> des <code>CameraCaptureSession.StateCallback</code> -Objekts aufgerufen. Sie ist dafür zuständig, den Autofokus für den Video-Modus einzuschalten und mit der Methode <code>setRepeatingRequest</code> die Videoaufnahme zu starten.
<i>onImageAvailableListener</i>	Sobald ein Bild verfügbar ist, wird diese Funktion aufgerufen. Das Bild wird dann zusammen mit der Information, ob der Bildschirm gedreht wurde, an die <code>onImageAvailable</code> -Methode des JNI weitergeleitet. Danach wird noch zum Aktualisieren der Text- und Bildinformationen die JNI Methode <code>updateViewInfo</code> im UI-Thread ausgeführt.

<i>onPause</i>	Diese Funktion regelt, was geschehen soll, wenn die App pausiert wird, z. B. durch Drücken des Home-Buttons. Hier werden die Kameraaufnahme gestoppt und die <code>onPause</code> -Methode des C++ Teils aufgerufen.
<i>checkPermissionsIfNecessary</i>	In dieser Funktion werden die für die Ausführung benötigten Berechtigungen für die Kameranutzung überprüft und angefordert, falls sie nicht gegeben sind. Die Methode <code>onRequestPermissionsResult</code> verarbeitet die Entscheidung des Users.

Für die Kameraeinbindung das Java Camera2 API statt dem nativen Äquivalent zu wählen, begründe ich folgendermaßen:

Zum einen ist die Einbindung und Verwendung im Code unkomplizierter und besser online dokumentiert. Zum anderen unterstützt das native Kamerainterface bestimmte Legacy-Hardware nicht mehr, sodass die Applikation auf einigen Geräten später nicht einsetzbar wäre.

Die Möglichkeit, das OpenCV Camera Interface wie bei iOS zu nutzen, ist auf Android nur im Java Teil möglich, was eine weitere Dependency bedeutet hätte.

Die Aktualisierung der Text- und Bildinformationen wird in dem Java-Teil ausgeführt. In C++ war es nicht möglich, auf den UI-Thread zuzugreifen.

Die Kamera-Steuerung der iOS-Version wurde nicht portiert, da Android keine systemeigenen Funktionen für die Gestenerkennung zur Verfügung stellt. Stattdessen wurde die Darstellung so angepasst, dass der Kameraausschnitt die Position verfolgt.

### 5.2.3.2. JNI

Das Java Native Interface ist in der Regel immer in zwei Dateien aufgeteilt. Eine ist in Java geschrieben (`VinsJNI.java`) und eine in C++ (`nativelib.cpp`).

Wie bei einem normalen Interface sind in der Java-Klasse nur die Funktionssignaturen definiert, also die Namen und die Parameterliste. In der C++ Datei finden sich dann diese Funktionen wieder. Dort wird auch der Funktionskörper definiert.

Zu beachten ist, dass die Package-Bezeichnung der Java-Klasse in den Namen mit einfließt. Wichtig ist dabei auch, dass die Funktionssignaturen exakt übereinstimmen, sowohl die Funktionsnamen als auch die Parameterliste. Befindet sich in letzterer ein Fehler, werden die Effekte frühestens zur Laufzeit deutlich, da weder die Entwicklungsumgebung noch der Compiler einen Error oder eine Warnung ausgeben.

Im Folgenden sind die Methoden mit ihren Funktionen kurz zusammengefasst:

<i>init</i>	Legt ein neues ViewController-Objekt an und speichert es in einem globalen Pointer. Führt eine Testmethode durch, die prüft, ob der ViewController ansprechbar ist und Ausgaben loggen kann. Ruft die <code>viewDidLoad</code> -Methode des ViewControllers auf, die ähnliche Funktionen wie die äquivalente Methode des iOS-Projekts übernimmt.
-------------	--

<i>onImageAvailable</i>	Diese Funktion ist der Hauptteil der Schnittstelle. Sie wandelt das Input-Bild, das als Buffer im YUV-Format vorliegt, in ein RGBA-Bild um. Weiterhin rotiert sie es, damit es die gleiche Orientierung wie das Kamerabild der iOS-Kamera hat. Dann wird es zusammen mit dem Timestamp der Funktion <code>processImage</code> übergeben. Damit das darin generierte Ausgabebild dargestellt werden kann, muss es im Anschluss noch einmal konvertiert, gedreht, je nach Orientierung gespiegelt und zuletzt in den Buffer der <code>TextureView</code> geschrieben werden.
<i>updateViewInfo</i>	In dieser Funktion werden alle <code>TextView</code> s mit den entsprechenden String-Variablen des <code>ViewController</code> s bestückt. Außerdem wird die Sichtbarkeit des Bildes mit den Initialisierungsinstruktionen entsprechend dem Status des Algorithmus angepasst.
<i>onPause</i>	Diese Methode ruft die <code>imuStopUpdate</code> -Methode des <code>ViewControllers</code> auf, um die <code>SensorEvents</code> zu unterbrechen.
<i>onARSwitch</i> & <i>onLoopSwitch</i>	Diese beiden Methoden geben die Information weiter, dass der jeweilige Schalter betätigt wurde. Im C++ Teil wird dann entsprechend darauf reagiert, indem das AR-Overlay und die Loop-Closure-Funktionen aktiviert werden.

### 5.2.3.3. C++

Da der Großteil des Objective C Codes in der `ViewController`-Klasse liegt, wurde versucht, so viel Code wie möglich daraus zu übernehmen. Daher wurde auch der Klassenname übernommen und die Implementierung auf die Header-Datei `ViewController.hpp` und die Source-Datei `ViewController.cpp` aufgeteilt. Die folgenden Bestandteile des Programms mussten geändert werden, da sie mit dem iOS SDK zusammenhängen oder aus anderen Gründen nicht kompilierbar waren:

<i>Logging</i>	Das Loggen der Debug-Informationen in die Standardausgabe ist in dem iOS Projekt über die <code>printf</code> - und <code>cout</code> -Funktionen gewährleistet. Unter Android müssen dafür die Logging-Funktionen des NDKs benutzt werden.
<i>NSThread</i>	VINS nutzt die iOS-Thread-Klasse <code>NSThread</code> . Sie wurde ersetzt durch die <code>Thread</code> -Klasse der Standard-Bibliothek. Die zusätzlichen Funktionen, die diese Klasse nicht bietet, wurden durch zusätzliche <code>Mutex</code> -Variablen ermöglicht. Zu diesen gehören das Abbrechen der Ausführung sowie weitere Synchronisierungsfunktionen.



<i>UIImage</i>	Die an vielen Stellen zum temporären Speichern sowie zum Anzeigen des Outputs verwendete Klasse <code>UIImage</code> wurde durch die OpenCV-Klasse <code>Mat</code> ersetzt und alle Operationen wurden entsprechend angepasst.
<i>UILabel</i>	Die UI-Elemente konnten nicht direkt aus dem C++ Teil manipuliert werden, da das nur in dem UI-Thread möglich ist. Aus diesem Grund werden alle Textausgaben in Strings zwischengespeichert und dann von außen per Polling verwendet.
<i>NSTimeInterval</i>	Unter iOS handelt es sich hierbei um einen Datentyp, der eine Zeitspanne in Sekunden angibt. Ersetzt bzw. weiterverwendet wurde er, indem er per <code>typedef</code> als <code>double</code> definiert wurde.
<i>IMU</i>	Um IMU-Sensordaten verarbeiten zu können, muss ähnlich wie für die Kamerainputs ein separater Looperthread angelegt werden. Diesem können dann EventQueues und Callback-Funktionen übergeben werden. In diesen Callback-Funktionen kann dann auf die Events der Queue zugegriffen werden.

### 5.2.4. Vergleich zwischen iOS und Android

In diesem Teil geht es um die Gemeinsamkeiten und Unterschiede von iOS und Android, die bei der Portierung deutlich wurden.

In vielen Dingen sind sich Android und iOS ziemlich ähnlich. Dazu zählen zum Beispiel der Zugriff auf die Sensor-Inputs der Kamera oder der IMU über Callback-Funktionen. Weitere Beispiele sind die Manipulation der Display-Texte, die nur aus dem UI-Thread möglich ist, oder die Beschreibung des UI-Layouts mittels deskriptiver XML-Sprachen in den `Main.storyboard` respektive `activity_main.xml`-Dateien.

Ein Unterschied zwischen den beiden Systemen liegt in der Komplexität des Codes der erforderlich ist, um die gewünschte Funktionalität umzusetzen. Die Übersichtlichkeit ist unter Android, gerade was das Programmieren in nativem Code mit den NDK-Klassen und -Funktionen betrifft, oft durch große Mengen an Boilerplate-Code belastet. In iOS ist zum Beispiel das Zugreifen auf die IMU-Daten über die `CMotionManager`-Klasse in wenigen Zeilen Code möglich, während in NDK-Code neben einem `SensorManager` noch ein `Looper` und ein `Queue`-Objekt separat angelegt und mit Parametern konfiguriert werden müssen.

Dieser Eindruck entstand, obwohl Objective C eine mir unbekannte Sprache war und die Syntax teilweise sehr ungewohnt sein kann, wenn man, einen C- / Java-Hintergrund hat. Die NDK-Funktionen wirken oft eher wie eine nachträglich hinzugefügte Schnittstelle zu den vorhandenen Java-Äquivalenten.

Von den Differenzen in der Software abgesehen, gibt es auch bei der Hardware teilweise entscheidende Unterschiede. So ist beispielsweise die Synchronisation der Sensor Event Timestamps, der inertialen Messeinheit und der Kamera auf Apple-Hardware

zuverlässiger und genauer. Die Probleme werden nun im folgenden Abschnitt 5.2.5 ausführlicher dargestellt.

### 5.2.5. Probleme

Bei der Portierung sind in der Programmierung Probleme aufgetreten, die teilweise auf die Android-Umgebung und teilweise auf das VINS-Mobile-Framework zurückzuführen sind.

Das erste größere Problem war und ist, dass der Objective C-Code des VINS-Frameworks an einigen Stellen nur spärlich dokumentiert ist und nicht immer ohne nähere Analyse klar wird, warum etwas getan wird. Erschwerend kommt hinzu, dass oft `public` Instanz-Variablen genutzt werden, um zwischen den Objekten zu kommunizieren. An anderen Stellen findet man auskommentierte Code-Fragmente oder teilweise ganze nicht verwendete Klassen.

Doch auch die Dokumentation des Android NDK ließ zu wünschen übrig, da in der offiziellen Online-Dokumentation nur die automatisch aus den Header-Kommentaren generierten Erklärungen der einzelnen Code-Objekte zu finden sind. Für die Verwendung wichtige Beispiele zur Nutzung der Klassen, Strukturen und Funktionen fehlen. Zusammen mit der großen Menge komplexen Boilerplate-Codes führt das zu einer Ungewissheit, ob der gewählte Weg der richtige ist oder wichtige Funktionsaufrufe vergessen wurden.

Auf der Hardware-Seite kam es, wie in dem vorherigen Abschnitt erwähnt, zu Problemen mit der Synchronisation der Sensor-Events. Bei Testausgaben der Sensor-Events auf Android konnte beobachtet werden, dass sich die Zeitdifferenz zwischen den Time-stamps der Kamera und der IMU während der Ausführung ohne ersichtlichen Grund um 300 ms verschieben kann. Dabei ist zu beachten, dass das Kamerabildintervall bei 30 Hz  $33.\bar{3}$  ms und das IMU-Messungsintervall bei 100 Hz gerade einmal 10 ms beträgt. Eine andere mögliche Erklärung für diese extreme Beobachtung ist, dass der Kamerabild-Buffer aufgrund steigender Prozessoranforderungen nicht mehr schnell genug abgearbeitet werden kann.

Außerdem ist nicht gewährleistet, dass bei gleicher Frequenz der Beschleunigungs- und Gyroskop-Events diese immer abwechselnd eintreffen. Es kann z. B. dazu kommen, dass erst 10 Beschleunigungs-Events und danach erst die 10 Gyroskop-Events auftreten, die eigentlich zeitlich parallel verarbeitet werden sollten. Daher war es notwendig das Buffer-System der IMU-Messungen neu zu schreiben.

Auch ist es infolge der Verarbeitungsreihenfolge von Kamera und IMU-Daten wichtig, dass die Bilder der Kamera zwischen dem Aufnahmezeitpunkt und dem Zeitpunkt des Bearbeitens einen größeren Abstand haben als die Messung der IMU. Ist das nicht der Fall, gehen einige Messungen der IMU verloren, weil sie noch nicht in die Queue eingefügt wurden, bevor das neueste Bild bearbeitet wurde. Für das nächste Bild werden sie vernachlässigt, weil ihr Timestamp noch vor dem des letzten Bildes liegt.

Während der ersten Tests kam es zu extremen Performance-Problemen der Ceres-Bibliothek und des VINS-Codes. Sie waren so signifikant, dass es nicht möglich war,

den Initialisierungsvorgang abzuschließen, da der Ceres-Solver jedes Mal vor dem Finden einer Lösung die maximal zulässige Solver-Zeit überschritten hatte.

Anfangs waren zwischen den in Li et al. (2017) angegebenen Zeitstatistiken der einzelnen Schritte und den gemessenen Werten Unterschiede in der Größenordnung des 20-Fachen und mehr zu beobachten. Genaue Zahlen dazu sind in Kapitel 6.2 zu finden.

Die einfachste Lösung war es, den Compiler von `clang` auf `gcc` zu wechseln, den Build-Modus auf Release festzulegen und das Optimierungslevel auf 3 zu stellen.

Nach diesen Optimierungen besteht noch immer ein deutlicher Performance-Unterschied, jedoch wird jetzt das Abschließen des Initialisierungsvorgangs dadurch nicht mehr verhindert.

### 5.3. Kalibrierung

Der Algorithmus setzt voraus, dass einige Hardwareparameter im Voraus bekannt sind. Zu diesen Parametern gehören die intrinsischen Kameraparameter Brennweite und Bildmittelpunkt und die extrinsischen Parameter, die die Transformation vom IMU- / Body-Frame in den Camera-Frame bestimmen. Die Hardware, auf der der Android-Port getestet werden sollte, war das Samsung Galaxy S7.

In Unterabschnitt 5.3.1 wird das Vorgehen beschrieben, das verwendet wurde, um die intrinsischen Parameter zu ermitteln. Unterabschnitt 5.3.2 schildert die Untersuchung der extrinsischen Parameter.

#### 5.3.1. Intrinsische Parameter

Die von dem Framework benötigten intrinsischen Parameter sind die Brennweite (FocalLength) und Bildmittelpunkt (PrincipalPoint), beide in Pixeln sowohl in X- als auch in Y-Richtung.

Verwendet wurde eine Kalibrierungsmethode mit einem 2D Schachbrettmuster. Zunächst wurde mit dem Smartphone ein Video gemacht, in dem das Muster aus verschiedenen Richtungen betrachtet wurde. Als Auflösung wurde 480 x 640 Pixel gewählt, da das Framework diese verwendet und sie für die korrekte Bestimmung der Brennweite in Pixeln übereinstimmen muss.

Aus dem Video wurden dann manuell mit dem Programm ShotCut<sup>5</sup> 15 bis 20 Frames extrahiert. Dabei wurde beachtet, dass nicht Positionen durch mehrere Frames abgedeckt wurden, sich also die Frames unterscheiden.

Im nächsten Schritt wurde die Software Suite MatLab<sup>6</sup> mit der Applikation *Camera Calibrator* genutzt, um die Parameter zu berechnen. Dazu müssen zunächst 10 bis 20 der Bilder zum Importieren ausgewählt werden. Daraufhin wird die Größe eines Schachbrettquadrats abgefragt. Wurde die Größe eingegeben, so besteht im folgenden Schritt die Möglichkeit, in einer Übersicht der importierten Bilder zu prüfen, ob in allen Bildern das Muster korrekt erkannt wurde und der Ursprung in der gleichen Ecke liegt.

---

<sup>5</sup><https://shotcut.org/>

<sup>6</sup><https://de.mathworks.com/products/matlab.html>

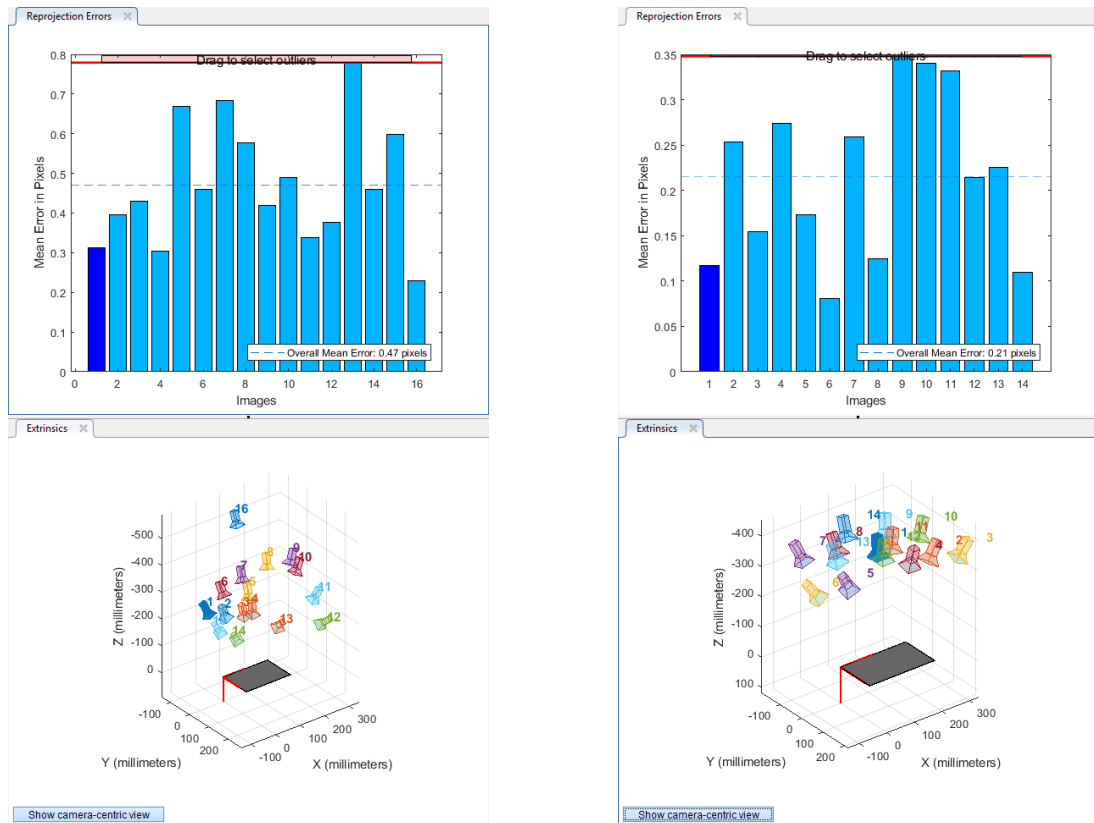
## 5. Portierung

Im letzten Schritt kann die eigentliche Kalibrierung gestartet werden. Als Resultat werden zunächst einmal zwei nützliche Informationen geliefert: Anhand einer Visualisierung der berechneten extrinsischen Parameter, also der Kamerapositionen relativ zu dem Schachbrettmuster, besteht noch einmal die Möglichkeit zu kontrollieren, ob alle Positionen den erwarteten Positionen entsprechen. Durch eine Diagramm-Visualisierung des Mittelwerts des Reprojection Error lassen sich auch Ausreißer einfach auswählen und entfernen.

Wenn das Ergebnis zufriedenstellend ist, können über die Schaltfläche *Export Camera Parameters* die gesuchten Parameter in das Hauptfenster von MatLab exportiert werden, wo sie dann in dem *Command Window* automatisch ausgegeben werden. Außer den benötigten Parametern **FocalLength** und **PrincipalPoint** werden auch andere Parameter ermittelt. Zu diesen zählen z. B. die Radiale Verzerrung (**RadialDistortion**), die aber in diesem Framework nicht beachtet wird.

Verwendet wurde ein Schachbrettmuster, das eine ungerade Anzahl an Reihen und eine gerade Anzahl an Spalten besitzt.

Anders als die meisten anderen Muster dieser Art, die man im Internet findet, ist bei dieser Kombination der Reihen- und Spaltenanzahl die Orientierung immer eindeutig. Dadurch kann MatLab beim Platzieren des Muster-Ursprungs nicht die falsche Ecke auswählen.



(a) Gedrucktes Schachbrettmuster

(b) Bildschirm-Schachbrettmuster

Abbildung 5.3.: Mean Reprojection Errors und Extrinsische Parameter

## 5. Portierung

Der erste Versuch wurde mit einem auf einem Din-A4-Blatt ausgedruckten Schachbrettmuster unternommen. Die einzelnen Quadrate hatten eine Seitenlänge von genau 30 mm. Da das Papier durch die große Menge an Druckertinte allerdings leicht unebene Wellen schlug, war der Reprojection Error relativ groß.

Deswegen wurde noch ein zweiter Versuch unternommen, bei dem das Muster auf dem Computerbildschirm angezeigt wurde. Die Größe der Quadrate war bei diesem Versuch mit 33.8 mm nur geringfügig größer.

Ergebnisse der Kalibrierung:

Tabelle 5.5.: Camera Intrinsics Printed Checkerboard

FocalLength:	[498.3953 495.7647]
PrincipalPoint:	[226.4976 319.1671]
RadialDistortion:	[0.2526 -0.5535]
ImageSize:	[640 480]
MeanReprojectionError:	0.4699

Tabelle 5.6.: Camera Intrinsics Screen Checkerboard

FocalLength:	[478.7620 478.6281]
PrincipalPoint:	[231.9420 319.5268]
RadialDistortion:	[0.2962 -0.6360]
ImageSize:	[640 480]
MeanReprojectionError:	0.2147

### 5.3.2. Extrinsische Parameter

Die von dem Algorithmus benötigten extrinsischen Parameter betreffen die relative Positionierung von der inertialen Messeinheit zu der Kamera. Gesucht war also die Translation IMU bzw. Body-Frame zu Kamera-Frame für das Testgerät Samsung Galaxy S7. Da aus dem Code nicht klar ersichtlich war, welches Koordinatensystem in welcher Geräteorientierung die Parameter des Frameworks benutzen, musste außerdem auch ein Abgleich mit der Position der IMU des iPhone 7 Plus gemacht werden.

In der Regel sind diese Informationen für die Entwicklung normaler Smartphone-Applikationen nicht von Relevanz, daher ist es nicht verwunderlich, dass die Hersteller die gesuchten Parameter nicht veröffentlicht haben.

Eine direkte Suche im Internet endete erfolglos, da auch Quellen, die sich genauer mit der Hardware der Smartphones auseinandersetzen, diese Informationen nicht angegeben haben.

Deshalb recherchierte ich anhand der im Internet gegebenen Bild- und Textinformationen, wo sich im Gerät die IMU befindet. Mit den bekannten Außenabmessungen und frontal aufgenommenen Fotos der Geräte konnte die genaue Position der IMU bestimmt werden. Dazu geeignet waren für das iPhone 7 Plus eine Röntgenaufnahme (iFixit, 2016a) und für das Galaxy S7 eine Aufnahme des geöffneten Geräts mit sichtbarer Leiterplatte (iFixit, 2016b).

## 5. Portierung

Um die Distanzen korrekt bestimmen zu können, mussten die Bilder der Geräte zunächst entlang dem Pixel-Raster ausgerichtet werden. Im nächsten Schritt wurden sie zur Vereinfachung der Messung so skaliert, dass genau 20 Pixel einem Millimeter entsprachen. Für diese Operationen wurden die frei verfügbaren Bildverarbeitungsprogramme GIMP<sup>7</sup> und Paint.NET<sup>8</sup> verwendet.

Eine Visualisierung der Resultate kann in Abbildung 5.4 betrachtet werden. Abgebildet sind auf der linken Seite das iPhone 7 Plus und auf der rechten Seite das Samsung Galaxy S7. Die Ansicht ist von vorne auf den Bildschirm blickend. In Grün sind die Messungen aus der oben beschriebenen Methode markiert. Blau kennzeichnet die Werte, die im Framework für das iPhone 7 Plus eingetragen sind.

Die Translation in Blickrichtung der Kamera wird vernachlässigt, da sie nicht ohne Weiteres feststellbar ist und ohnehin sehr gering ausfällt.

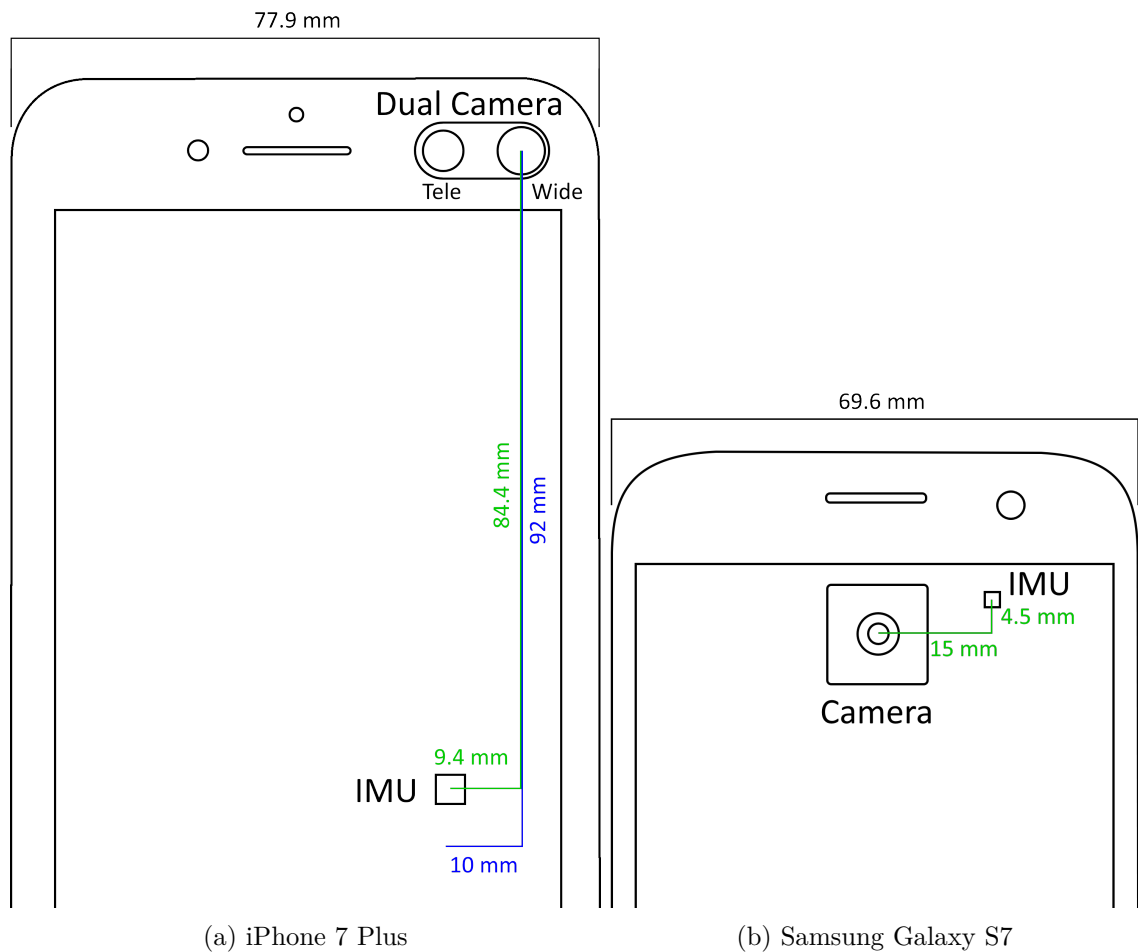


Abbildung 5.4.: Extrinsische Parameter: Translation IMU zu Kamera

<sup>7</sup><https://www.gimp.org/>

<sup>8</sup><https://www.getpaint.net/>

## 6. Ergebnisse

Nach Portierung des Frameworks habe ich dieses mit der originalen iOS-Version in Performance, Robustheit und Genauigkeit verglichen. Verwendet wurde dazu als Test-Hardware das Android-Smartphone Samsung Galaxy S7. Die genauen Hardware-Spezifikationen sind in Abschnitt 6.1 aufgeführt.

Die Messergebnisse der quantitativen Performance-Untersuchung werden in Abschnitt 6.2 vorgestellt.

Zuletzt behandelt Abschnitt 6.3 die qualitativen Unterschiede in Form einer experimentellen Untersuchung. Als weiterer Vergleichsgegenstand wurde mit Google Tango ein Hardware-gestütztes AR-Framework hinzugezogen.

### 6.1. Test-Hardware

Die relevanten Hardware-Spezifikationen des Testgeräts Samsung Galaxy S7 sind die Folgenden:

Prozessor:	Exynos 8890 64 Bit Octa-core (4x2.3 GHz & 4x1.6 GHz)
Arbeitsspeicher:	4 GB
Kamera:	12 MP Frontkamera (f/1.7, 26mm) mit optischer Bildstabilisierung

Tabelle 6.1.: Samsung Galaxy S7 (GSMArena.com, [n. d.]c)

Damit lässt es sich der mittleren bis oberen Leistungs-Klasse zuordnen.

In Abschnitt 6.3 werden außerdem das Apple iPad Pro und das Lenovo Phab 2 Pro hinzugezogen:

Prozessor:	Apple A9X 64 Bit Dual-core (2.26 GHz)
Arbeitsspeicher:	4 GB
Kamera:	8 MP Frontkamera (f/2.4, 31mm)

Tabelle 6.2.: Apple iPad Pro 12.9 (2015) (GSMArena.com, [n. d.]a)

Prozessor:	Qualcomm Snapdragon 652 64 Bit Octa-core (4x1.8 GHz & 4x1.4)
Arbeitsspeicher:	4 GB
Kamera:	16 MP Frontkamera, Global-Shutter Weitwinkel-Kamera für das Motion Tracking, Time-Of-Flight Tiefen-Kamera
Alle Sensoren sind auf Hardware-Ebene synchronisiert.	

Tabelle 6.3.: Lenovo Phab 2 Pro (GSMArena.com, [n. d.]b)

Das iPad Pro und das Lenovo Phab 2 Pro waren Leihgaben der Technischen Hochschule. Leider stand kein iPhone zur Verfügung, wie es in (Li et al., 2017) zum Durchführen der Tests verwendet wurde.

## 6.2. Performance auf Android Geräten

In diesem Abschnitt werden die statistischen Ergebnisse des Performance Benchmarks präsentiert.

Ermittelt wurden diese Werte anhand der schon in dem Projekt-Code befindlichen Logging-Ausgaben. Einige weitere wurden hinzugefügt.

Zur Beobachtung der vergangenen Prozessor-Zeit verwendet VINS die OpenCV Funktion `cv::getTickCount()`. Verpackt ist sie zur einfacheren Verwendung in die beiden Makros `TS(Bezeichner)` und `TE(Bezeichner)` zum Starten und Beenden des Timers.

Nach geringer Modifikation schreibt dieser Benchmark-Timer die Messergebnisse in das Android-Logging-System *logcat*. Der Output kann nach Tags gefiltert und für die spätere Verarbeitung in eine Textdatei umgeleitet werden.

Aus diesem Text können dann im Nachhinein mittels regulärer Ausdrücke die gewünschten Information extrahiert werden. Verwendet habe ich dazu das Programm Notepad++<sup>1</sup>.

Für die weitere Verwendung und statistische Auswertung habe ich die Daten in das Tabellenkalkulationsprogramm Excel<sup>2</sup> eingefügt. Mit wenig Aufwand ist es dort möglich, Säulen-, Boxplot- und andere Verteilungsdiagramme für die Auswertung zu erstellen.

In Tabelle 6.4 ist eine Übersicht der statistischen Werte zu sehen.

<sup>1</sup><https://notepad-plus-plus.org/>

<sup>2</sup><https://products.office.com/de-de/excel>



## 6. Ergebnisse

Thread	Modul	Freq. (Hz)	iOS	vor Opt.	nach Opt.
1	Feature Detection	10	17 ms	14 ms	16 ms
	Feature Tracking	30	3 ms	8 ms	8 ms
	Visualisierung	30		-	6 ms
2	Initialisierung	once	80 ms	>2060 ms	268 ms
	Nonlinear Optimization	10	60 ms	-	113 ms
3	Loop Detection	3	60 ms	-	34 ms
	Feature Retrieval	3	10 ms	-	44 ms
4	Global Pose Optimization	marg.		-	29 ms

Tabelle 6.4.: Übersicht Performance-Vergleich

Die Struktur und die Werte für iOS sind aus Li et al. (2017) entnommen. Die beiden rechten Spalten beschreiben die Werte für Android vor und nach der Optimierung, die in Abschnitt 5.2.5 beschrieben wurde.

### 6.2.1. Vor Compiler-Optimierungen

Für diese Messung habe ich einen Testlauf von 50 Sekunden durchgeführt. Über diesen Zeitraum wurden die Mittelwerte, die oben in der Tabelle abgebildet sind, berechnet. Die genauen Zusammensetzungen der Daten finden sich im Anhang A.1.

Vor den Compiler-Optimierungen des VINS-Programmcodes und der Ceres-Bibliothek kam es zu so großen Performance-Problemen, dass einige Programmteile gar nicht zur Ausführung kamen. Die Initialisierung konnte nie abgeschlossen werden, da schon bei dem ersten Einsatz der Ceres-Bibliothek für die Vision-Only SfM-Berechnung (Abschnitt 4.3.1) das Zeitlimit für den Solver überschritten wurde und der Vorgang deshalb abgebrochen wurde. Dabei sei angemerkt, dass der Grenzwert standardmäßig bei 0.6 Sekunden lag und schon erhöht war.

Der beobachtete Wert von 8 ms für das Feature Tracking bestätigt meine Vermutung, dass OpenCV für iOS besser optimiert ist. Die beobachtete Zeit für die Feature-Detection kann von Versuch zu Versuch schwanken. Sie steht im direkten Zusammenhang mit der Qualität des Kamerabildes und der Bewegungsgeschwindigkeit. Je mehr Features im Tracking verloren gehen, desto mehr müssen neu gefunden werden und desto höher ist auch der Zeitaufwand. Daher kann der bessere Wert im Vergleich mit iOS auch im Zusammenhang mit den Umgebungsbedingungen stehen.

Die Visualisierungs-Zeit in der Initialisierungsphase in die Tabelle zu übernehmen macht keinen Sinn, da sie dieser Phase extrem gering ist. Die nicht lineare Optimierung und alle Loop Closure-Programmteile wurden nicht erreicht.

### 6.2.2. Nach Compiler-Optimierungen

Für die Messungen der optimierten Version habe ich zwei verschiedene Ansätze gewählt.

Zum einen habe ich 10 Durchläufe ausgeführt, die ich direkt nach der Initialisierung abgebrochen habe. Sie dienten alleine der Informationsgewinnung über die Initialisie-

## 6. Ergebnisse

rungszeiten. Da diese in der Regel nur einmal pro Test auftreten, wurde diese Anzahl der Durchläufe gewählt, um einen aussagekräftigen Mittelwert zu bekommen.

Zum anderen habe ich noch 5 weitere Testläufe durchgeführt, die jeweils auf 20 Sekunden begrenzt waren. In allen wurde die Initialisierung abgeschlossen und in 4 von 5 auch erfolgreich Loops erkannt und geschlossen. Die genauen Daten dieser Versuche befinden sich im Anhang A.2.

In den Zeiten der Initialisierung ist der Unterschied der Optimierung ganz deutlich zu beobachten. Auch wenn der zeitliche Grenzwert des Solvers bei 2 Sekunden lag, gab es in den Messungen Ausreißer, die 5 Sekunden überschritten und auch nicht zu einer Lösung geführt haben. Mit der Lösung des SFM-Optimierungs-Problems ist der Initialisierungsvorgang aber noch nicht beendet. Danach würde noch das Visual-Inertial Alignment folgen, das in den 268 ms der optimierten Version enthalten ist. Das lässt nur den Schluss zu, dass irgendetwas Ceres mit den alten Compiler-Optionen extrem gebremst hat. Es könnte sich dabei um Debug-Funktionen, wie z.B. assertions o.Ä. handeln.

In den ersten zwei Zeilen ist gut zu sehen, dass Funktionen, die vor allem auf OpenCV aufbauen, keinen Unterschied aufweisen, da OpenCV als externe vorkompilierte Bibliothek von den Änderungen nicht betroffen ist. Zu diesen Funktionen zählt die Feature-Detection und das Feature-Tracking.

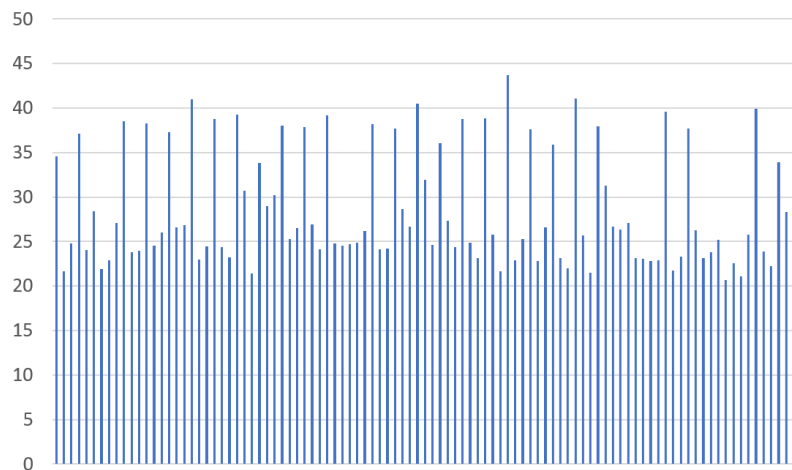


Abbildung 6.1.: Zeitstatistik `ViewController::processImage()`

Nicht in der Tabelle 6.4 angegeben ist die Gesamtzeit der `processImage`-Funktion. Sie führt sowohl die Feature-Detection als auch das Tracking aus. In Abbildung 6.1 ist diese Messung als Säulendiagramm veranschaulicht. Dabei fällt auf, dass die Feature-Detection mit 10 Hz sich nicht gleichmäßig verteilt, sondern in jedem dritten Frame Ausschläge in der Frame-Zeit bildet. Auch wenn die Zeit mit 29.25 ms im Schnitt oberhalb der 30 Hz Grenze liegt, führt es zu ungleichmäßigen Input-Zeiten. Ein ausreichend großer Input-Buffer kann diese Schwankungen ausgleichen. Es kommt allerdings hinzu, dass die gesamte `onImageAvailable`-Funktion des JNIs im Schnitt 37.25 ms beansprucht. Diese Funktion umfasst zusätzlich auch die Konvertierungen des Kamerabildes. Dadurch kann die Bild-Frequenz von 30 Hz nicht eingehalten werden, sodass

einige Kamera-Frames übersprungen werden. Die entstehende Verzögerung und das Überspringen der Frames ist mit dem bloßen Auge bemerkbar, betrifft aber teilweise auch die iOS-Version.

Schuld an diesem fast doppelt so hohen Zeitanspruch sind die verschiedenen Bildkonvertierungen in OpenCV. Neben dem unvermeidlichen Farbraum-Wechsel von YUV zu RGB sind vor allem die `cv::rotate`-Operationen für 90°-Rotationen sehr rechenintensiv. Jede dieser Rotation beansprucht alleine 3 – 5 ms. Diese Operation muss mehrmals pro Frame durchgeführt werden, da das Framework nur eine Auflösung von 480px × 640px annimmt, die Android-Kamera aber im Landscape-Modus nur 640px × 480px hergibt.

Auch die Visualisierung nach der Initialisierungsphase kostet Prozessorzeit. Der Unterschied zwischen den beiden Phasen wird in Anhang A.2 gut veranschaulicht.

Die beobachteten Werte für die Loop-Detection und die Global Pose Optimization sind wider Erwarten relativ gering. Das lässt sich aber auf die kurze Testdauer zurückführen. Wächst im Verlauf der Zeit der Keyframe-Graph immer weiter an, so wird auch diese Funktion mehr Zeit beanspruchen.

Insgesamt bestätigen die Ergebnisse meine Vermutung, dass das Zusammenspiel zwischen Software und Hardware auf iOS besser optimiert ist. Wären die Bibliotheken besser parallelisiert, so würden die Ergebnisse aufgrund der höheren Multithreading-Performance Android favorisieren.

### 6.3. Experimentelle Ergebnisse

In diesem Abschnitt werden die Ergebnisse des experimentellen Vergleichs zwischen der Android-Portierung, der iOS-Version und Google Tango als weiter Augmented Reality-Technologie vorgestellt.

#### 6.3.1. Versuchsbedingungen

Dieser Unterabschnitt behandelt die Überlegungen zur Vergleichbarkeit der Systeme und die gewählte Lösung.

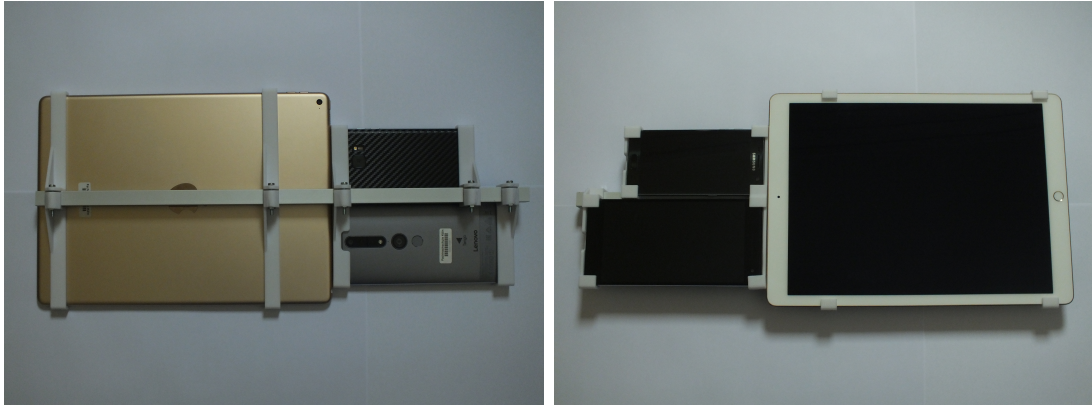
In diesem Projekt ging es vor allem darum, die Fähigkeiten der mobilen Plattformen zu vergleichen. Die Plattformen sind aber teilweise sehr verschieden, vor allem in den Sensor-Inputs, die direkten Einfluss auf die Qualität und Performance des Frameworks haben. Daher macht es wenig Sinn, ein öffentlich verfügbares Dataset, wie z.B. MH\_03\_medium in (Li et al., 2017) o.ä., für den Vergleich zu nutzen.

Stattdessen wurde ein gleichzeitiger Life-Test gewählt, bei dem die unterschiedlichen Plattformen den gleichen Umgebungsbedingungen ausgesetzt werden. Praktisch ist es natürlich physikalisch nicht möglich, die exakt gleichen Bedingungen zu erzeugen, da die Kameras nicht alle an der gleichen Position sein können.

Um trotzdem möglichst ähnliche Bedingungen zu gewährleisten, habe ich eine Testapparatur entworfen, an der die Geräte fest fixiert werden können. Dabei habe ich auch darauf geachtet, dass die Kameras möglichst frei und gleichzeitig nahe beieinander liegen. Außerdem sollten die Touchscreens frei zugänglich bleiben, um eine problemlose Bedienung zu ermöglichen. Für das Design der Halterungen habe ich das

## 6. Ergebnisse

CAD-Programm Fusion 360<sup>3</sup> verwendet. Ziel dabei war es, neben den oben genannten Bedingungen ein simples Design zu wählen, das im anschließenden 3D-Druck-Vorgang möglichst wenig Filament braucht und einfach druckbar ist. Deshalb wurde eine Verbundkonstruktion mit einem Aluminium Vierkantrohr gewählt. Abbildung 6.2 zeigt die Konstruktion mit den eingespannten Geräten.



(a) Rückansicht

(b) Frontansicht

Abbildung 6.2.: Testkonstruktion

Die Durchführung der Versuche wurde aufgeteilt in Indoor und Outdoor, da einige äußere Faktoren in den beiden Umgebungen unterschiedlich stark ausgeprägt sind. Zu diesen zählen die unterschiedlich großen Entfernungen zu den beobachteten Features, die unterschiedlich große zurückgelegte Entfernung und andere visuelle Faktoren wie die Lichteinstrahlung. Sie können eine wichtige Rolle für die Qualität der Odometrie spielen.

In beiden Fällen wurden die Bildschirme des Android und des Google Tango-Geräts aufgenommen. Da es unter der iOS-Version 10 nicht mehr ohne weiteres möglich ist, dies zu tun, habe ich auf dem iPad an wichtigen Punkten Bildschirmfotos gemacht. Die Videos sind auf dem beigelegten Datenträger zu finden (Anhang B).

Im Folgenden werden die Ergebnisse der Odometrie anhand von Screenshots vorgestellt und ausgewertet.

### 6.3.2. Indoor Experiment

Dieser Unterabschnitt behandelt den Indoor-Test. Es werden die Vorgehensweise und die Ergebnisse des Tests vorgestellt.

Für den Indoor-Test habe ich als Ausgangspunkt einen Schaukasten gewählt, der sich von der sonst recht repetitiven Umgebung der Gänge im Gebäude abhob. Zu Beginn habe ich die Geräte eine Zeit lang auf den Schaukasten gerichtet, damit möglichst viele Features erkannt und gespeichert werden. Als nächstes habe ich dann einer L-Form folgend eine frei liegende Brücke überquert. Trotz der vielen Streben und reflektierenden Fensterflächen konnte das Tracking gut aufrecht erhalten werden. Am Ende der

<sup>3</sup><https://www.autodesk.de/products/fusion-360/overview>

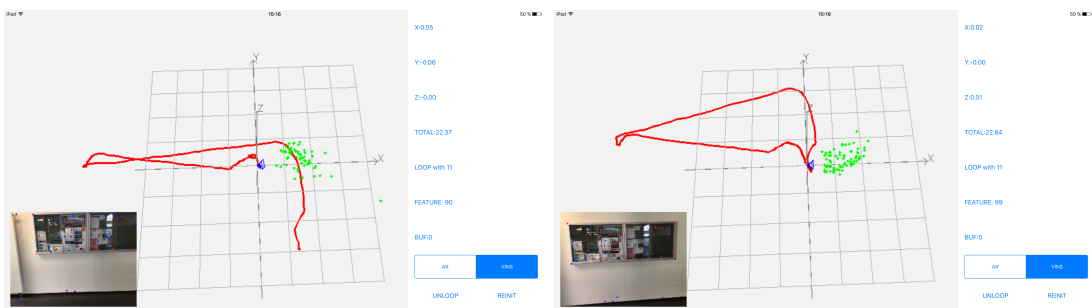
## 6. Ergebnisse

Bücke habe ich mich um  $180^\circ$  gedreht und den selben Weg zurück genommen. Vor dem Schaukasten angekommen richtete ich die Geräte wieder darauf.

Sowohl die iOS-Version als auch die Android-Portierung waren in der Lage, den Ort wieder zu erkennen und ihre Pose, die inzwischen reichlich Drift angesammelt hatte, zu korrigieren. In den Screenshots in Abbildung 6.3 und 6.4 ist diese Loop Closure festgehalten.

Das Google Tango-Gerät lief im reinen Motion Tracking-Modus, es hatte also keine Möglichkeit, Loops zu erkennen. Trotzdem konnte es die Bewegung sehr viel präziser verfolgen und unterlag nicht dem Drift, der bei beiden VINS-Systemen aufgetreten ist. Die erzeugte Laufbahn ist in Abbildung 6.5 zu sehen.

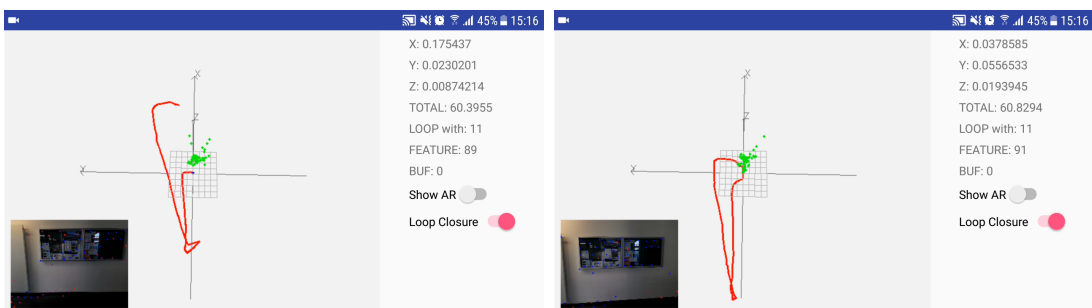
Nach dem Zurückkehren an den Ausgangspunkt habe ich die Prozedur noch einmal wiederholt. Diesmal war die iOS-Version wieder in der Lage, den Schaukasten zu erkennen. Die Android-Version hat es auch nach einigen Sekunden Wartezeit nicht geschafft. Der Test wurde daraufhin beendet.



(a) Rückkehr zum Ausgangspunkt

(b) Loop Closure gefunden

Abbildung 6.3.: iOS Indoor-Test



(a) Rückkehr zum Ausgangspunkt

(b) Loop Closure gefunden

Abbildung 6.4.: Android Indoor-Test

## 6. Ergebnisse

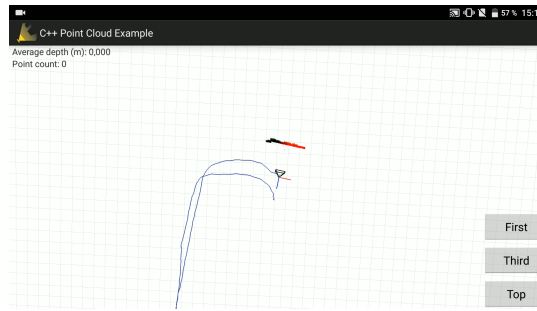


Abbildung 6.5.: Google Tango-Test

### 6.3.3. Outdoor Experiment

Der Outdoor-Test wurde ähnlich gestaltet wie der Indoor-Test. Zu Beginn habe ich ein markantes Objekt fokussiert und es eine Weile betrachtet. In diesem Fall war es aber kein in greifbarer Nähe befindliches, sondern ein Pavillon auf dem Campus-Gelände. Anders als im Indoor-Test wurde für diesen Test eine Laufbahn gewählt, die nicht auf dem gleichen Weg hin und zurück führt, sondern eher einem Rechteck gleicht.

Während des Test-Verlaufs war zu beobachten, dass außen das Tracking wesentlich stabiler und präziser war als im Indoor-Test. Zurückzuführen ist das auf die Feature reichere Umgebung und die besseren Lichtverhältnisse. Beide Geräte konnten mit den starken Helligkeitsschwankungen durch den direkten Sonnenschein gut zurecht kommen. Android war insgesamt präziser, was das reine Tracking betrifft. Geschuldet ist das vermutlich der Hardware des iPads. Die Kamera ist von geringerer Bild-Qualität und sie hat häufige Autofokusprobleme.

Nachdem ich zu dem Ausgangspunkt zurückgekehrt war, waren die Positionen von Android und Google Tango beide sehr nahe an der realen Position, während die iOS-Version mit größerem Drift zu kämpfen hatte. Bei dem anschließenden Betrachten des Pavillons ist es allerdings auch hier Android nicht gelungen, sich zu relokalisieren, während iOS damit keine Probleme hatte. Dies entsprach nicht meiner Erwartung, dass eine Feature-reichere Umgebung auch eine bessere Loop-Erkennung ermöglicht.

Der Test wurde daraufhin abgeschlossen. In den Abbildungen 6.6 und 6.7 kann der finale Status begutachtet werden.

## 6. Ergebnisse

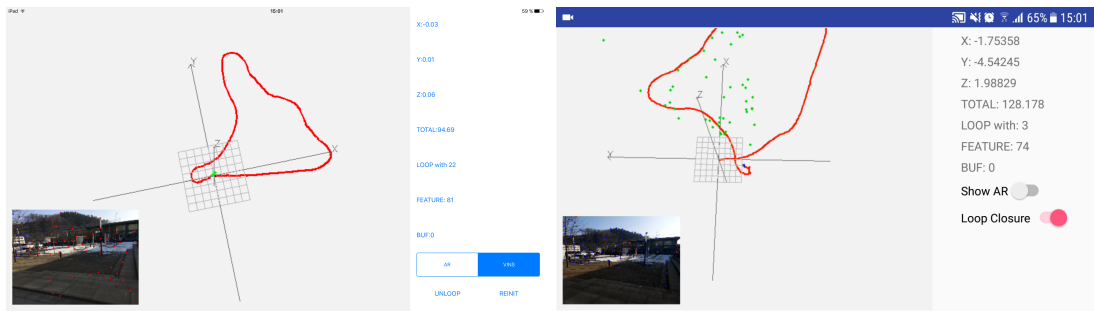


Abbildung 6.6.: iOS vs Android Outdoor-Test

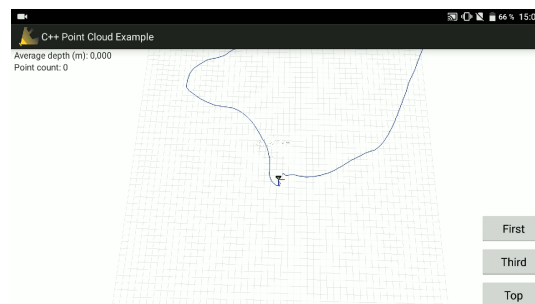


Abbildung 6.7.: Google Tango Outdoor-Test

## 7. Diskussion

In diesem Kapitel werden die Ergebnisse aus Kapitel 6 noch einmal aufgegriffen und kritisch bewertet. Ich beschäftige mich mit der Fragestellung, ob das Framework VINS-Mobile für die Verwendung in dem Projekt *Augmented Reality Campus* der TH Köln in Frage kommt. Dazu erläutere ich die Bedingungen, die für den späteren Zielkontext erfüllt sein müssen, und gebe eine Einschätzung, ob diese erreichbar sind. Außerdem zähle ich die minimalen Schritte auf, die nötig sind, um das Framework im Kontext mit der Unity-Engine zu verwenden.

### 7.1. Ergebnis-Bewertung

In Abschnitt 6.2 tritt die erste Schwäche der portierten Android-Version zu Tage: die Performance. Während die höheren Zeitkosten der nicht linearen Optimierung keine Probleme zu machen scheinen, führt die Überbelastung des Kamera-Threads zu einer merklichen Input-Verzögerung. Ein solcher Effekt kann in einer Augmented Reality-Anwendung in der Praxis vom Nutzer als äußerst unangenehm empfunden werden.

In Abschnitt 6.3 wird die zweite Schwäche deutlich: die Loop Detection. Hier müsste noch eine genauere Analyse durchgeführt werden, welche Ursachen für die Differenzen zwischen Android und iOS verantwortlich sind.

Verbessert werden könnte auch die Testmethode. In der aktuellen Konfiguration ist die Ergebnis-Auswertung beschränkt auf die visuelle Auswertung des Positionsverlaufs. Es fehlt eine Art „Ground-Truth“, anhand derer der Error quantifiziert werden kann. Dafür wäre zusätzliche Hardware nötig, die diese liefert. Google Tango ist für diese Aufgabe ungeeignet, da auch dieses System je nach Umgebung Drift unterworfen ist.

### 7.2. Anforderungen durch AR-Campus

Um die Lösung effektiv in der Praxis nutzen zu können, müssen einige Anforderungen erfüllt sein:

#### ***Generelle Lauffähigkeit auf verschiedensten Android-Smartphones***

Android hat neben den Unterschieden in Software, auf die ich in Abschnitt 5.2.4 eingegangen bin, im Gegensatz zu iOS eine viel größere Bandbreite an Hardware, die unterstützt werden muss.

Bei der Portierung habe ich bereits beispielsweise bei der Wahl der Kamera-API darauf geachtet, eine möglichst große Bandbreite an Hardware abzudecken. Trotzdem muss damit gerechnet werden, dass das Framework nicht ohne Probleme auf allen Android-Geräten laufen wird.



Ein weiteres Problem stellt die Kalibrierung dar. Sie hat maßgeblichen Einfluss auf die Qualität des Trackings. Während im aktuellen Zustand die Kalibrierung manuell und separat durchgeführt werden muss, ist das im späteren Kontext mit vielen verschiedenen Smartphones keine Option mehr.

Dort sollte eine automatisierte Kalibrierungsmethode angestrebt werden.

### ***Stabilität des Trackings und der Anwendung***

Für die Immersion und Nutzbarkeit der AR-Anwendung ist die Stabilität des Trackings essentiell. Bei temporärer Verdeckung der Kamera oder zu schnellen Bewegungen kommt es schnell zu Tracking-Verlust. Die Wiederherstellung des korrekten Zustandes über die Loop Detection weist auf Android noch Schwächen auf, die es zu beheben gilt.

In den Tests konnte ich außerdem beobachten, dass die VINS-Bibliothek intern mindestens eine Stelle hat, an der von Zeit zu Zeit ein Adressierungs-Fehler mit entsprechender Exception auftritt. Das hat den Absturz der Anwendung zur Folge. Einmal kam es sogar dazu, dass während des Test das Smartphone komplett abgestürzt ist und erst nach Verbinden mit meinem Computer wieder neu gestartet werden konnte. Solche schwerwiegenden Fehler dürfen im praktischen Einsatz bei dem Nutzer nicht auftreten.

### ***Geringe Performance-Anforderung, um Kapazitäten für AR-Content zu sparen und die Batterie zu schonen***

In der Endanwendung sollte der Tracking-Algorithmus eher eine untergeordnete Rolle in der Prozessor-Beanspruchung spielen, damit genügend Ressourcen zur Verfügung stehen, um 3D-Objekte zu rendern. Aktuell ist die Prozessorbelastung noch sehr hoch. Das hat auch negative Auswirkungen auf die Batterielaufzeit. Dieses Problem besteht allerdings zur Zeit bei allen Augmented Reality-Frameworks. Sogar bei Google Tango, welches dedizierte Hardware für die Tango-Funktionalitäten nutzt, ist dies ein Problem.

Wie in Kapitel 6 festgestellt, besteht Optimierungsbedarf in der Pipeline der Bildverarbeitung. Einfache Verbesserungen wären die Eliminierung der vielen Bild-Transformationen. Auch das Neu-Kompilieren des OpenCV SDKs mit einem höheren Optimierungslevel könnte Abhilfe schaffen. Da der Erfolg aber ungewiss und der Vorgang sehr zeitaufwändig ist, wurde in dieser Arbeit davon abgesehen.

### ***Möglichkeit zur Nutzung der Umgebungs-Wiedererkennung mit vorher gespeicherten Daten (Markern) und Erzeugung dieser***

Aktuell unterstützt VINS nur die Wiedererkennung von Orten aus dem selben Testlauf. Um das Lokalisierungs-System effektiv nutzen zu können, müssen zwei Bedingungen erfüllt werden. Zum einen muss die Möglichkeit bestehen, vorgefertigte Daten einzuspeisen. Statt der klassischen 2D-Marker würde es sich hierbei um eine Art 3D-Objekt-Information handeln. Zum anderen muss es auch für die Ersteller von Inhalten einfach möglich sein, neue Informationen anzulegen und in das System einzuspeisen.

### ***Verwendung mit der Unity-Engine***

Die angestrebte Entwicklungs-Plattform ist die Unity-Engine. Sie bietet Cross-Plattform-Kompatibilität für viele Systeme, unter anderem auch iOS und Android. Um

## 7. Diskussion

VINS dort nutzen zu können, muss ein Android-Java-Plugin implementiert werden, das dann in Unity eingebunden wird. Das gleiche gilt für iOS.

Wünschenswert wäre es, wenn das VINS-Framework von der Anwendungsentwicklung getrennt als Blackbox verwendet werden könnte. Der Input würde die Kalibrierungs-Parameter und „Marker“-Informationen und der Output die Tracking-Daten umfassen.

Da es durchaus möglich ist, dass in Zukunft die geplante Entwicklungsumgebung geändert wird, wäre es von Vorteil, das Plugin so modular wie möglich zu gestalten. Zusätzlich könnte ein Refactoring der Architektur der VINS-Implementation helfen, um die Übersichtlichkeit und Verständlichkeit zu verbessern. Ein Negativ-Beispiel dafür ist die „Gottklasse“ `ViewController`.

## 8. Zusammenfassung und Ausblick

Ziel dieser Arbeit war es, das Framework VINS Mobile (Li et al., 2017) von der iOS-Plattform auf die Android-Plattform zu portieren und das Ergebnis auf Eignung im Kontext des Projekt Augmented Reality Campus zu analysieren.

Im ersten Teil dieser Arbeit wurde in Kapitel 4 der dem Framework zugrunde liegende Algorithmus im Detail vorgestellt.

In einem praktischen Teil habe ich das Framework portiert und an die Nutzung auf Android angepasst. Die Details zu dem Vorgehen und den Ergebnissen habe ich im zweiten Teil der Arbeit in Kapitel 5 beschrieben. Zuletzt habe ich im dritten Teil der Arbeit das Ergebnis der Portierung in Kapitel 6 und 7 untersucht und diskutiert.

Dabei bin ich zu dem Schluss gekommen, dass die Android-Portierung in Funktionsumfang und Qualität mit der iOS-Version fast gleichwertig ist. Bis zur Nutzbarkeit im Projekt AR-Campus sind aber noch einige Arbeitsschritte nötig.

Neben der in Kapitel 7 angesprochenen möglichen Weiterentwicklung der Portierung gibt es auch für zukünftige Forschungsprojekte interessante, an diese Arbeit anknüpfende Themen.

Eines davon ist, die Eignung des „Bag-of-Words“-Ansatzes für markerlose Objekt- bzw. Umgebungswiedererkennung zu untersuchen. Mit DBoW3<sup>1</sup> und FBOW<sup>2</sup> stehen dafür zwei Weiterentwicklungen der DBoW2-Implementation zur Wahl, die Geschwindigkeitsoptimierungen und weniger externe Bibliotheks-Abhängigkeiten versprechen.

Um einen möglichst ressourcensparenden Betrieb zu ermöglichen, könnte zusätzlich eine GPS-Anbindung implementiert werden. Sie würde die Lokalisierung durch eine Eingrenzung der Suchmenge aller möglicher Positionen unterstützen und so die Prozessor- und Speicherbelastung senken.

Ein anderes anspruchsvolles Thema ist die Gewinnung von dichten Tiefen-Informationen aus den Kamerabildern. Im Kontext des AR-Campus-Projekts würde sie die Verdeckung von AR-Objekten durch reale Objekte ermöglichen und so die Immersion steigern.

---

<sup>1</sup><https://github.com/rmsalinas/DBoW3>

<sup>2</sup><https://github.com/rmsalinas/fbow>

## 9. Persönliche Entwicklung

In diesem Kapitel reflektiere ich meine Erfahrungen, die ich aus dieser Abschlussarbeit gewinnen konnte. Ich gehe auf die Dinge ein, die ich retrospektiv betrachtet anders angegangen und gelöst hätte und ziehe Schlüsse für zukünftige Arbeiten.

Diese Abschlussarbeit stellt meinen ersten Kontakt mit der Entwicklungsumgebung iOS dar. Auch wenn ihre Nutzung ein unerheblicher Teil der Arbeit war, konnte ich einen interessanten Einblick in das Ecosystem gewinnen und mir einen groben Eindruck verschaffen.

Ebenso konnte ich zwei neue Bibliotheken (Ceres und DBoW2) kennenlernen und habe zum ersten Mal OpenCV in einer so stark hardware-beschränkten Umgebung eingesetzt. Daneben konnte ich mir im Verlauf der Arbeit viel neues fachliches Wissen im Bereich der Computer Vision aneignen.

In Bereichen, mit denen ich bereits vertraut war, konnte ich meine Kenntnisse weiter ausbauen. Dazu gehört vor allem der gesamte Setup-Prozess mit dem Einbinden verschiedenster Codequellen in CMake. Schon in meiner Praxisprojektarbeit habe ich CMake recht gut kennengelernt, aber wie sich in dieser Bachelorarbeit herausstellte, ist das Build-System wesentlich komplexer, komplizierter und undurchsichtiger, als es mir bisher erschien. Dementsprechend war der Zeitaufwand für den Praxisteil der Arbeit deutlich größer als erwartet und hat meinen ursprünglichen Zeitplan gesprengt.

Deutlich ist mir dadurch geworden, wie wichtig die Aufrechterhaltung der Motivation und Zuversicht angesichts eines unmöglich erscheinenden Ziels für die eigene Leistungsfähigkeit ist.

Außerdem habe ich die Erfahrung machen müssen, dass die für meine Arbeit notwendigen APIs und Bibliotheken zum Teil schlecht dokumentiert waren und dadurch den Erfolg meines Projektes stark gefährdeten. In Zukunft werde ich daher größeres Augenmerk darauf richten, in welchem Zustand ebensolche Frameworks und APIs sind, welchen Aufwand es kostet, sie in das eigene Projekt einzubinden, und das bei meiner Evaluation stärker einbringen.

Im Nachhinein hätte ich eine andere Bearbeitungsreihenfolge gewählt. Die Ausformulierung der Algorithmus-Erklärung würde ich an den Anfang meiner Arbeit stellen. Das hätte mir vermutlich bei dem Verständnis der Implementierung geholfen und Zeit gespart.

Zuletzt ist die gewählte Sprache zu überdenken. Natürlich ist mir Deutsch als meine Muttersprache geläufiger als Englisch. Doch im Laufe meiner Arbeit habe ich festgestellt, dass viele englischen Fachbegriffe keine geläufige deutsche Entsprechung besitzen. Das Verfassen einer Arbeit zu dieser Thematik macht deshalb in Englisch mehr Sinn.

Abschließend betrachtet war diese Bachelorarbeit eine neue große Herausforderung, die mein Fachwissen entscheidend erweitert hat und mich in organisatorischer Planung und im Umgang mit Widerständen persönlich gestärkt hat.

# Abbildungsverzeichnis

4.1.	Blockdiagramm der gesamten Pipeline mit Zusammenhängen der einzelnen Bestandteile (nach Li et al., 2017, S. 2) . . . . .	4
4.2.	Sliding Window Formulation (Li et al., 2017, S. 3) . . . . .	5
4.3.	Visual Inertial Alignment (Li et al., 2017, S. 5) . . . . .	9
4.4.	Marginalization Strategy (Qin et al., 2017, S. 8) . . . . .	14
4.5.	Loop Closure (Qin et al., 2017, S. 10) . . . . .	15
5.1.	CMake Hierarchie . . . . .	20
5.2.	Vergleich der User Interfaces . . . . .	28
5.3.	Mean Reprojection Errors und Extrinsische Parameter . . . . .	35
5.4.	Extrinsische Parameter: Translation IMU zu Kamera . . . . .	37
6.1.	Zeitstatistik <code>ViewController::processImage()</code> . . . . .	41
6.2.	Testkonstruktion . . . . .	43
6.3.	iOS Indoor-Test . . . . .	44
6.4.	Android Indoor-Test . . . . .	44
6.5.	Google Tango-Test . . . . .	45
6.6.	iOS vs Android Outdoor-Test . . . . .	46
6.7.	Google Tango Outdoor-Test . . . . .	46

# Tabellenverzeichnis

5.5. Camera Intrinsic Printed Checkerboard . . . . .	36
5.6. Camera Intrinsic Screen Checkerboard . . . . .	36
6.1. Samsung Galaxy S7 (GSMArena.com, [n. d.]c) . . . . .	38
6.2. Apple iPad Pro 12.9 (2015) (GSMArena.com, [n. d.]a) . . . . .	38
6.3. Lenovo Phab 2 Pro (GSMArena.com, [n. d.]b) . . . . .	39
6.4. Übersicht Performance-Vergleich . . . . .	40

# Literaturverzeichnis

- Sameer Agarwal, Keir Mierle, and Others. [n. d.]. Ceres Solver. <http://ceres-solver.org>. ([n. d.]).
- Michael Calonder, Vincent Lepetit, Christoph Strecha, and Pascal Fua. 2010. Brief: Binary robust independent elementary features. In *European conference on computer vision*. Springer, 778–792.
- D. Gálvez-López and J. D. Tardós. 2012. Bags of Binary Words for Fast Place Recognition in Image Sequences. *IEEE Transactions on Robotics* 28, 5 (2012), 1188–1197. <https://doi.org/10.1109/TR0.2012.2197158>
- GSMArena.com. [n. d.]a. Apple iPad Pro 12.9 (2015) - Full tablet specifications. ([n. d.]). Retrieved 20.02.2018 from [https://www.gsmarena.com/apple\\_ipad\\_pro\\_12\\_9\\_\(2015\)-7562.php](https://www.gsmarena.com/apple_ipad_pro_12_9_(2015)-7562.php)
- GSMArena.com. [n. d.]b. Lenovo Phab2 Pro - Full phone specifications. ([n. d.]). Retrieved 20.02.2018 from [https://www.gsmarena.com/lenovo\\_phab2\\_pro-8145.php](https://www.gsmarena.com/lenovo_phab2_pro-8145.php)
- GSMArena.com. [n. d.]c. Samsung Galaxy S7 - Full phone specifications. ([n. d.]). Retrieved 20.02.2018 from [https://www.gsmarena.com/samsung\\_galaxy\\_s7-7821.php#g930f](https://www.gsmarena.com/samsung_galaxy_s7-7821.php#g930f)
- iFixit. 2016a. iPhone 7 Plus Teardown. (Sept. 2016). Retrieved 20.02.2018 from <https://de.ifixit.com/Teardown/iPhone+7+Plus+Teardown/67384#s136470>
- iFixit. 2016b. Samsung Galaxy S7 Teardown. (March 2016). Retrieved 20.02.2018 from <https://de.ifixit.com/Teardown/Samsung+Galaxy+S7+Teardown/56686#s122920>
- Peiliang Li, Tong Qin, Botao Hu, Fengyuan Zhu, and Shaojie Shen. 2017. Monocular Visual-Inertial State Estimation for Mobile Augmented Reality. In *2017 IEEE International Symposium on Mixed and Augmented Reality*, Wolfgang Broll, Holger Regenbrecht, and J. Edward Swan (Eds.). IEEE, Piscataway, NJ, 11–21. <https://doi.org/10.1109/ISMAR.2017.18>
- D. Nister. 2004. An efficient solution to the five-point relative pose problem. *IEEE Transactions on Pattern Analysis and Machine Intelligence* 26, 6 (June 2004), 756–770. <https://doi.org/10.1109/TPAMI.2004.17>
- Tong Qin, Peiliang Li, and Shaojie Shen. 2017. VINS-Mono: A Robust and Versatile Monocular Visual-Inertial State Estimator. (2017). <http://arxiv.org/pdf/1708.03852>

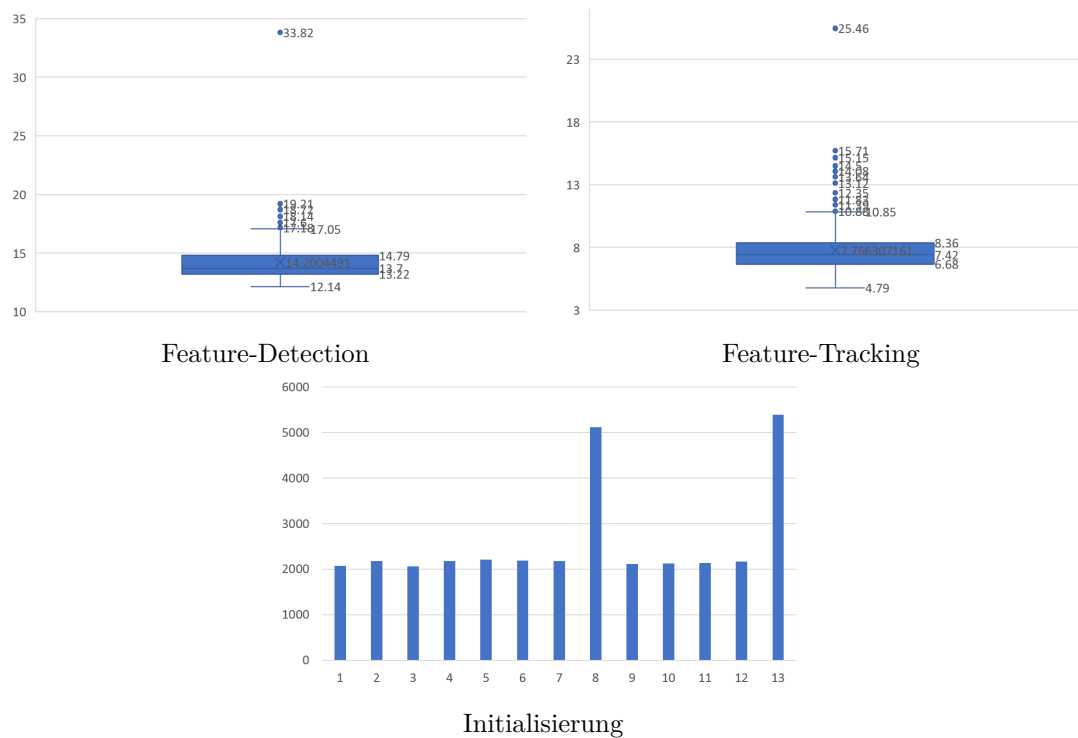
Edward Rosten and Tom Drummond. 2006. Machine learning for high-speed corner detection. In *European conference on computer vision*. Springer, 430–443.



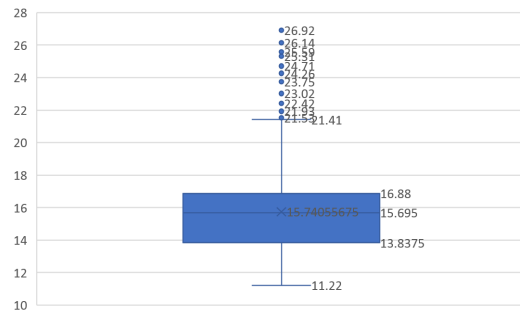
# Anhang A. Performance Statistiken

Die Y-Achse beschreibt die Zeit in ms, die X-Achse bei den Säulen-Diagrammen die Nummer der Messung.

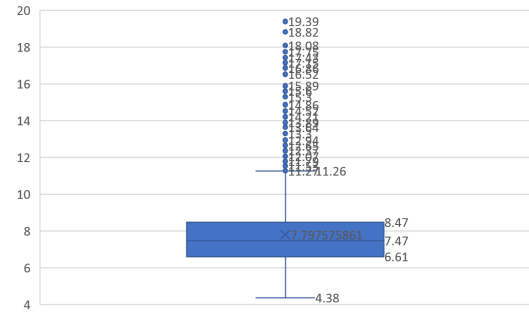
## A.1. Vor Compiler-Optimierung



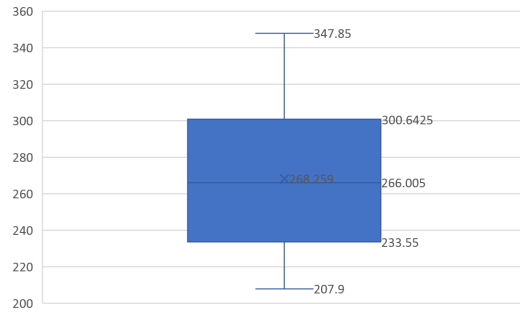
## A.2. Nach Compiler-Optimierung



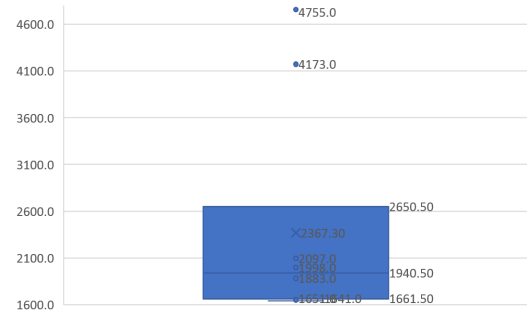
Feature-Detection



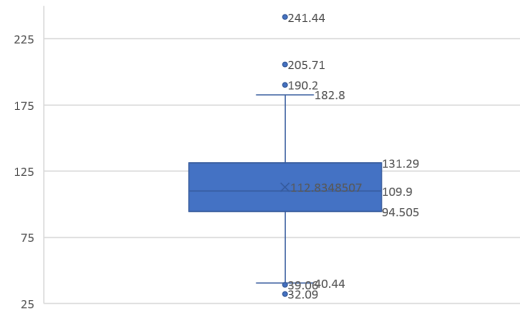
## Anhang A. Performance Statistiken



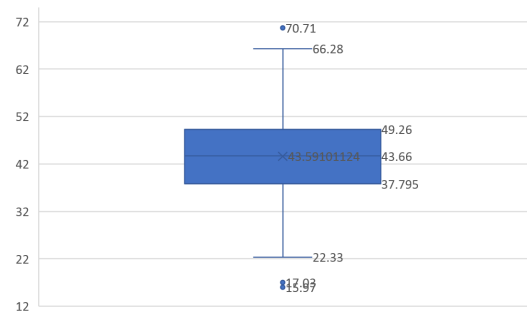
Initialisierung



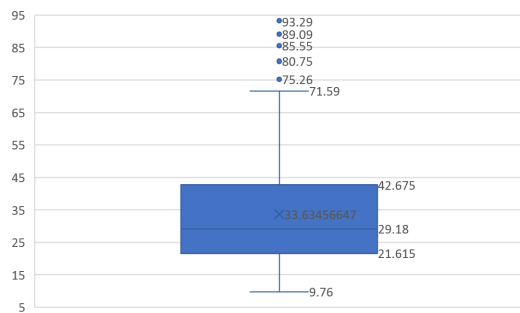
App-Start bis Ende der Initialisierung



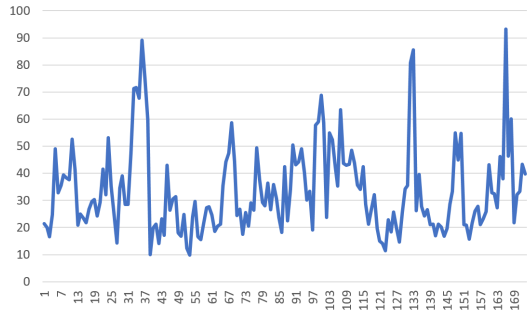
Non-Linear Optimization



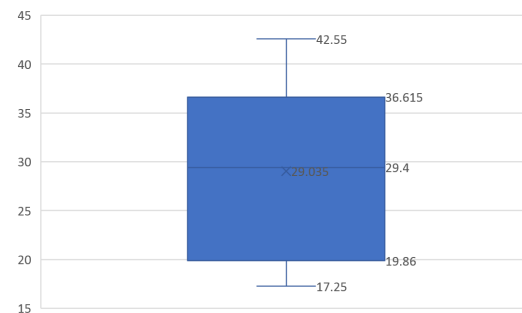
Feature-Retrieval



Loop Detection



Loop Detection Verlauf



Global Pose Optimization

## Anhang B. Digitale Anlagen

Die digitalen Anlagen auf dem beigelegten Datenträger umfassen folgende Dateien:

- Android Studio Projekt mit dem Quellcode und den kompilierten Bibliotheken
- Videos der Experimente
- Excel-Tabellen der Performance-Untersuchung
- archivierte Internetquellen
- digitale Kopie dieser Arbeit

# Eidesstattliche Erklärung

Ich versichere, die von mir vorgelegte Arbeit selbständig verfasst zu haben.

Alle Stellen, die wörtlich oder sinngemäß aus veröffentlichten oder nicht veröffentlichten Arbeiten anderer entnommen sind, habe ich als entnommen kenntlich gemacht. Sämtliche Quellen und Hilfsmittel, die ich für die Arbeit benutzt habe, sind angegeben.

Die Arbeit hat mit gleichem Inhalt bzw. in wesentlichen Teilen noch keiner anderen Prüfungsbehörde vorgelegen.

Gummersbach, 23. Februar 2018

Jannis Malte Möller